Elaborazione del Linguaggio Naturale con Python

Steven Bird, Edward Lopez, Erwin Klein

Traduzione italiana a cura di S.Lariccia

Con il contributo di:

Federica Savini, Annagiulia Scaini, Gabriela Primicerio, Andrea Placidi, Elettra Raffaela Melucci

1 L’elaborazione del linguaggio e Phyton

È facile acquisire milioni di parole di testo. Cosa possiamo farne, presupponendo di saper scrivere alcuni semplici programmi? In questo capitolo ci concentreremo sulle seguenti domande:

1. Cosa possiamo ottenere combinando semplici tecniche di programmazione con ampie quantità di testo?
2. Come possiamo estrarre automaticamente parole-chiave e frasi che riassumano lo stile ed il contenuto di un testo?
3. Quali strumenti e tecniche ci fornisce per tale lavoro il linguaggio di programmazione Python?

Questo capitolo è suddiviso in sezioni che alternano due stili abbastanza diversi. Nelle sezioni “Quantificare con il linguaggio” ci occuperemo di alcuni processi di programmazione motivati linguisticamente senza necessariamente spiegare come funzionino. Nelle sezioni “Approfondimento su Python” esamineremo in maniera sistematica i concetti chiave di programmazione. Segnaleremo i due stili nei titoli delle sezioni, ma più avanti i capitoli misceleranno entrambi gli stili senza specificarlo preventivamente. Ci auguriamo che questo stile d’introduzione vi dia un assaggio genuino di quello che ci aspetta in seguito mentre si assolvono una serie di concetti elementari di linguistica ed informatica. Chi abbia una familiarità di base in entrambi i campi può saltare alla Sezione 1.5; ripeteremo i punti più importanti nei capitoli a seguire, e nel caso abbiate perso qualcosa potete facilmente consultare il materiale di riferimento on line all’indirizzo http://www.nltk.org/. Nel caso in cui il materiale vi fosse totalmente nuovo, questo capitolo solleverà più domande che risposte, domande di cui ci occuperemo nel resto del libro.

1.1 Quantificare con il Linguaggio: Testi e Parole

Conosciamo tutti il testo, in quanto lo leggiamo e scriviamo ogni giorno. In questo paragrafo considereremo il testo come *dati grezzi* per i programmi che scriviamo, programmi che lo manipolano ed analizzano in una varietà di modi interessanti. Ma prima di poterlo fare dobbiamo iniziare ad usare l’interprete Python.

Iniziare ad usare Python

Una delle cose più apprezzate di Python è che permette di digitare direttamente sul suo **interprete** interattivo ovvero il programma che gestisce i tuoi programmi Python. Si può accedere all’interprete Python utilizzando una semplice interfaccia grafica chiamata Sviluppo Ambientale Interattivo (in inglese Interactive DeveLopment Environment da

cui l'acronimo IDLE). Su Mac potete trovarlo sotto *Applicazioni→MacPython*, e su Windows sotto *Tutti i programmi→Python*. Sotto Unix potete avviare Python dalla struttura digitando idle (nel caso non fosse installato, provate a digitare python). L'interprete riporterà una striscia riguardo la vostra versione di Python; controllate semplicemente d'aver avviato Python 2.4 o 2.5 (questo è 2.5.1):

```
Python 2.5.1 (r251:54863, Apr 15 2008, 22:57:26)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Nota

Se non siete in grado di avviare l'interprete Python, probabilmente Python non è installato correttamente. Siete pregati di visitare il sito `http://python.org/` per istruzioni più dettagliate.

Il prompt `>>>` indica che l'interprete Python è ora in attesa dell'introduzione dei dati. Quando copiate esempi da questo libro, non digitate il "`>>>`". Ora, cominciamo, utilizzando Python come una calcolatrice:

```
>>> 1 + 5 * 2 - 3
8
>>>
```

Una vola che l'interprete ha terminato di calcolare la soluzione e la visualizza, il prompt riappare. Questo significa che l'interprete Python è in attesa di un'altra istruzione.

Nota

Il vostro turno: Inserite qualche altra espressione di vostra invenzione. Puoi usare l'asterisco (`*`) per moltiplicare e la barra (`/`) per dividere, e le parentesi per mettere tra parentesi le espressioni. Notate che la divisione non sempre si comporta come ci si potrebbe aspettare. Effettuate la divisione tra numeri interi (con arrotondamento di frazioni per difetto) quando digitate `1/3` e la divisione con il numero in "virgola mobile" (o decimale) quando digitate `1.0/3.0`. Per ottenere il comportamento di divisione previsto (standard in Python 3.0), dovete digitare: `from __future__ import division`

Gli esempi precedenti dimostrano come si possa lavorare interattivamente con l'interprete Python, sperimentando con varie espressioni di linguaggio per vedere cosa accade. Ora proviamo un'espressione illogica per vedere come se la cava l'interprete:

```
>>> 1 +
  File "<stdin>", line 1
    1 +
```

```
      ^
SyntaxError: invalid syntax
>>>
```

Questo ha prodotto un **errore sintattico**. In Python non ha senso terminare un'istruzione con un segno positivo. L'interprete Python indica il rigo in cui è avvenuto il problema (linea 1 di `<stdin>`, che sta per "standard input").

Ora che possiamo usare l'interprete Python, siamo pronti per iniziare a lavorare con i dati di linguaggio.

Iniziare ad usare NLTK

Prima di proseguire dovreste installare NLTK, scaricabile gratuitamente da `http://www.nltk.org/`. Seguite le istruzioni del sito per scaricare la versione richiesta dalla vostra piattaforma.

Una volta installato NLTK, avviate l'interprete Python come prima, ed installate i dati necessari per il libro digitando i seguenti due comandi sul prompt Python, dopodiché selezionate la collezione come mostrato nella Figura 1.1.

```
>>> import nltk
>>> nltk.download()
```

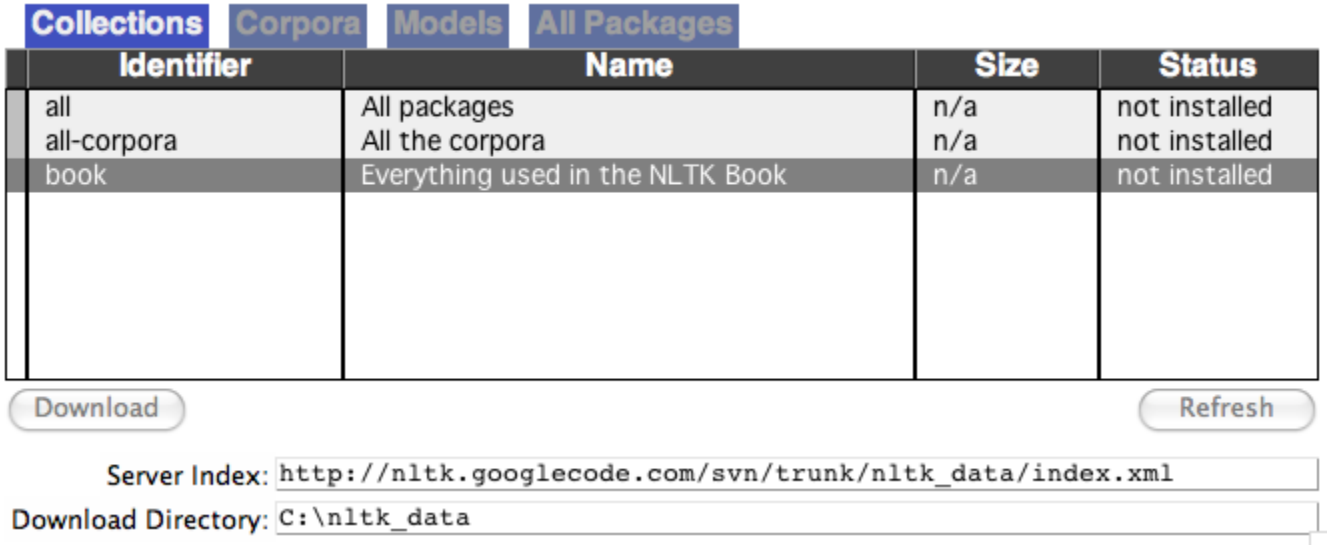


Figura 1.1: Scaricando la Raccolta di libri NLTK: scorrete i pacchetti disponibili usando `nltk.download()`. La tabella delle **Raccolte** sul downloader mostra come i pacchetti siano raggruppati in serie, dovreste selezionare la linea etichettata **book** per ottenere tutti i dati richiesti per gli esempi e gli esercizi del libro. Si tratta di circa 30 file compressi che necessitano di circa 100Mb di spazio sul disco. L'intera raccolta di dati (ovvero **tutti** quelli presenti sul downloader) è circa cinque volte questa misura (al momento in cui si scrive) e continua ad espandersi.

Una volta che I dati sono stati scaricati sul vostro apparecchio, potete caricarne un po' usando l'interprete Python. Il primo passo è digitare un comando speciale sul prompt Python che dice all'interprete di caricare una parte dei testi così da esplorarli: `from nltk.book import *`. Questo dice "dal modulo NLTK `book` , carica tutti gli oggetti." Il modulo contiene tutti i dati di cui avrai bisogno mentre leggete questo capitolo. Dopo aver inviato un messaggio di benvenuto, caricate il testo di diversi libri (ci impiega pochi secondi). Ecco di nuovo il comando, insieme al risultato che vedrete. Fate attenzione che lo spelling ed la punteggiatura siano corretti, e ricordate che non dovete digitare`>>>` .

```
>>> 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
>>>
```

Ogni volta che vogliamo esplorare tra questi testi, dobbiamo solo inserire i loro nomi nel prompt Python:

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

Ora che possiamo usare l'interprete Python, e abbiamo qualche dato con cui lavorare, siamo pronti per iniziare.

Cercare il testo

Ci sono molti modi per esaminare il contesto di un testo oltre che semplicemente leggendolo. Un esame delle concordanze ci mostra ogni occorrenza di una data parola, assieme ad un po' di contesto. Andiamo a cercare la parola mostruoso in *Moby Dick* inserendo **`text1`** seguito da un punto, poi il termine **`concordance`**, e poi inserendo `"monstrous"` tra parentesi.

```
>>> text1.concordance("monstrous")
Building index...
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
that has survived the flood ; most monstrous and most mountainous ! That
Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and
more de
th of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall
ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am
strongly
ere to enter upon those still more monstrous stories of them which are to
be fo
ght have been rummaged out of this monstrous cabinet there is no telling
. But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up
dead u
>>>
```

Nota

Il vostro turno: Provate a cercare altre parole; per evitare di digitare più volte, dovreste essere in grado di usare freccia in alto, Ctrl-freccia in alto o Alt-p per accedere a comandi precedenti e modificare la parola da cercare. Puoi anche provare a cercare su alcuni degli altri testi che abbiamo incluso. Per esempio, cerca in *Ragione e Sentimento* la parola *affetto*, usando `text2.concordance("affection")`. Esamina il libro della Genesi per scoprire quanto a lungo hanno vissuto alcune persone usando `text3.concordance("lived")`. Potresti guardare al `text4`, il *Corpus dei discorsi d'insediamento* per vedere alcuni esempi di inglese risalente al 1789 e andare alla ricerca di parole come *nazione, terrore* e *dio* per vedere come queste parole sono state impiegate in maniera differente nel tempo. Abbiamo incluso anche il `text5`, il *NPS Chat Corpus*: andate alla ricerca di parole anticonvenzionali come *im*, *ur*, *lol*. (Notate che questo corpus è incensurato!)

Una volta che avete trascorso un po' di tempo ad esaminare questi testi, ci auguriamo che abbiate un nuovo senso della ricchezza e della diversità del linguaggio. Nel prossimo capitolo imparerete come accedere ad una più ampia varietà di testo, includendo il testo in lingue altre dall'inglese.

Una concordanza ci permette di vedere le parole nel contesto. Ad esempio, abbiamo visto che *monstrous* si ritrova in contesti quali *the ___ pictures* e *the ___ size* . Quali altre parole appaiono in una simile varietà di contesti? Possiamo scoprirlo apponendo il termine `similar` al nome del testo in questione, poi inserendo la parola rilevante tra parentesi:

```
>>> text1.similar("monstrous")
Building word-context index...
subtly impalpable pitiable curious imperial perilous trustworthy
abundant untoward singular lamentable few maddens horrible loving lazy
mystifying christian exasperate puzzled
>>> text2.similar("monstrous")
Building word-context index...
very exceedingly so heartily a great good amazingly as sweet
remarkably extremely vast
>>>
```

Osservate come si possano ottenere risultati differenti da testi differenti. Austen usa questa parola diversamente da Melville; per lei, *monstrouos* ha connotazioni positive, e a volte funziona da intensificatore come la parola *very* (molto).

Il termine `common_contexts` ci consente di esaminare solo i contesti che vengono condivisi da due o più parole, come *monstrous* e *very*. Dobbiamo mettere queste parole sia tra parentesi quadre che parentesi tonde e separarle con una virgola:

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

Nota

Il vostro turno: Scegliete un altro paio di parole e comparate il loro utilizzo in due testi differenti usando le funzioni `similar()` e `common_contexts()`.

Una cosa è rintracciare automaticamente che una particolare parola ricorra in un testo e mostrare alcune parole che appaiono nello stesso contesto. Tuttavia, possiamo anche determinare la *posizione* di una parola in un testo: dopo quante parole dall'inizio appare. Quest'informazione di posizione può essere mostrata usando un **grafico di dispersione**. Ogni striscia rappresenta l'istanza di una parola, e ogni riga rappresenta l'intero testo. Nella Figura 1.2 possiamo vedere modelli impressionanti dell'uso di parole negli ultimi 220 anni (in un testo artificiale costruito unendo i testi del corpus dei Discorsi d'insediamento uno dietro l'altro). Potete produrre questa trama come mostrato sotto. Potrebbe interessarvi provare altre parole (ad esempio *libertà*, *costituzione*), e testi differenti. Potete prevedere la dispersione di una parola prima di vederla? Come prima, fate attenzione ad inserire correttamente le virgolette, le virgole, la parentesi quadre e tonde.

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

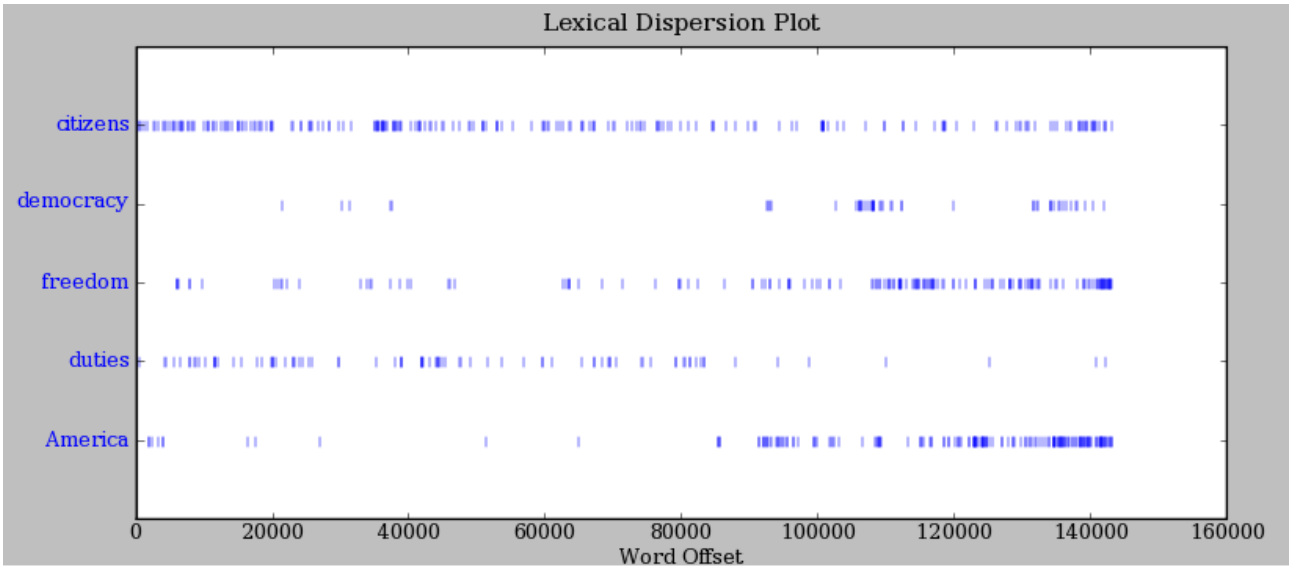


Figura 1.2: Grafico di dispersione lessicale di parole nei Discorsi d'Insediamento dei Presidenti americani. Può essere usato per indagare i cambiamenti dell'uso della lingua nel corso del tempo.

Nota

Importante: è necessario aver installato i pacchetti NumPy e Matplotlib di Python per creare i grafici usati in questo libro. Siete pregati di visitare il sito `http://www.nltk.org/` per istruzioni sull'installazione.

Adesso, solo per divertimento, proviamo a generare qualche testo casuale nei vari stili che abbiamo appena visto. Per farlo, digitiamo il nome del testo seguito dal termine `generate`. (Dobbiamo inserire le parentesi tonde ma senza nulla in mezzo.)

```
>>> text3.generate()
In the beginning of his brother is a hairy man , whose top may reach
unto heaven ; and ye shall sow the land of Egypt there was no bread in
all that he was taken out of the month , upon the earth . So shall thy
wages be ? And they made their father ; and Isaac was old , and kissed
him : and Laban with his cattle in the midst of the hands of Esau thy
first born , and Phichol the chief butler unto his son Isaac , she
>>>
```

Notate che la prima volta che avviate questo commando va lentamente perché raccoglie le statistiche delle sequenze di parole. Ogni volta che lo avvii otterrai come risultato un testo diverso. Ora proviamo a generare un testo casuale nello stile di un discorso d'insediamento o di una chat room di Internet. Nonostante il testo sia casuale riutilizza parole comuni e frasi del testo di provenienza e ci dà un senso del proprio stile e contenuto. (Cosa manca a questo testo generato casualmente?)

Nota

Quando produce il proprio risultato, la punteggiatura viene separata dal lavoro precedente. Nonostante si tratti di una formattazione scorretta per il testo inglese lo facciamo per evidenziare che le parole e la punteggiatura sono indipendenti l'una dall'altra. Apprenderete maggiormente a riguardo nel Capitolo 3.

Contare il vocabolario

Il fatto più ovvio sui testi che emerge dagli esempi precedenti è che differiscono nel vocabolario che utilizzano. In questa sezione vedremo come utilizzare il computer per contare le parole in un testo in una varietà di modi utili. Come prima, vi butterete e sperimenterete con l'interprete Python anche se non aveste ancora studiato Python sistematicamente. Mettete alla prova la vostra conoscenza modificando gli esempi e provando gli esercizi alla fine del capitolo.

Proviamo ad iniziare scoprendo la lunghezza di un testo dall'inizio alla fine, in termini di parole e simboli di punteggiatura che appaiono. Usiamo il termine `len` per ottenere la lunghezza di qualcosa, che qui applicheremo al libro della Genesi:

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

Quindi la Genesi ha 44,764 parole e simboli di punteggiatura o "gettoni." Un **gettone** è il nome tecnico per una sequenza di caratteri, come *capelli, suo*, o :), che vogliamo trattare come un gruppo. Quando contiamo il numero di gettoni in un testo, diciamo, la frase *to be or not to be*, stiamo contando le occorrenze di queste sequenze. Perciò, nel nostro esempio di frase ci sono due occorrenze di *to*, due di *be*, ed una sia di *or* che *not*. Ma ci sono solo quattro distinti vocaboli in questa frase. Quante parole distinte contiene il libro della Genesi? Per scoprirlo con Python dobbiamo porre la questione in maniera leggermente diversa. Il vocabolario di un testo è solo una *serie* di gettoni che si usa, in quanto in una serie, tutti i duplicati si ripiegano assieme. In Python possiamo ottenere i vocaboli del `text3` con il comando `set(text3)`. Quando lo facciamo molte schermate di parole scorreranno velocemente. Ora provate quanto segue:

```
>>> sorted(set(text3)) [1]
['!', "'", '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',
'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]

>>> len(set(text3)) [2]
2789
>>>
```

Aggiungendo `sorted()` all'espressione Python `set(text3)` [1], otteniamo una lista di vocaboli smistata, cominciando con vari simboli di punteggiatura e continuando con

parole che iniziano per *A*. tutte le parole in maiuscolo precedono le parole in minuscolo. Scopriamo la mole del vocabolario indirettamente, chiedendo il numero di oggetti in una serie, e inoltre, possiamo usare `len` per ottenere questo numero [2]. Nonostante abbia 44,764 gettoni, questo libro ha solo 2,789 parole distinte, o "tipi di parole." Un **tipo di parola** è la forma o la compitazione di una parola indipendente dalle sue specifiche occorrenze in un testo, ovvero la parola considerata come un vocabolo unico. Il nostro conto di 2,789 oggetti includerà simboli di punteggiatura, quindi chiameremo generalmente questi unici oggetti **tipi** invece di tipi di parola.

Adesso calcoliamo la misura della ricchezza lessicale di un testo. Il prossimo esempio ci mostra che ogni parola viene usata mediamente 16 volte (dobbiamo accertarci che Python usi la divisione con numero in virgola mobile):

```
>>> from __future__ import division
>>> len(text3) / len(set(text3))
16.050197203298673
>>>
```

Dopodiché concentriamoci su parole particolari. Possiamo contare quanto spesso una parola ricorre in un testo e calcolare quale percentuale di un testo è occupata da una parola specifica:

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

Nota

Il vostro turno: Quante volte appare la parola *lol* in `text5`? A quale percentuale corrisponde rispetto al numero totale di parole nel testo?

Potreste ripetere i calcoli su vari testi ma è noioso continuare a ripetere la formula. Potete, invece, inventare voi un nome per un compito, come "diversità lessicale" o "percentuale", ed associarlo ad una porzione di codice. Ora devi solo digitare un nome breve invece di una o più linee complete del codice Python, e poi riutilizzarlo quante volte ci piace. La porzione di codice che svolte un compito per noi è chiamata **funzione** ed assegniamo un nome breve alla nostra funzione con la parola chiave `def`. Il prossimo esempio mostra come definire due nuove funzioni, `lexical_diversity()` e `percentage()`:

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

```
...     return 100 * count / total
...
```

Attenzione!

L'interprete Python cambia il prompt da `>>>` a `...` dopo aver raggiunto i due punti alla fine della prima linea. Il prompt `...` indica che Python aspetta di veder apparire una **porzione di codice a capo**. Spetta a voi effettuare il rientro dal margine digitando quattro spazi o premendo il tasto TAB. Per terminare la porzione non rientrata basta inserire una linea vuota.

Nella definizione di diversità lessicale, specifichiamo un **parametro** nominato `text`. Questo parametro è un "segnaposto" per il vero testo del quale vogliamo calcolare la diversità lessicale, e ricorre nella porzione di codice che si avvierà quando viene usata la funzione. Similarmente `percentage()`, è preposto ad occuparsi di due parametri, nominati conto e totale.

Una volta che Python sa che `lexical_diversity()` e `percentage()` sono i nomi delle porzioni specifiche di codice, possiamo andare avanti ed usare queste funzioni:

```
>>> lexical_diversity(text3)
16.050197203298673
>>> lexical_diversity(text5)
7.4200461589185629
>>> percentage(4, 5)
80.0
>>> percentage(text4.count('a'), len(text4))
1.4643016433938312
>>>
```

Ricapitolando, possiamo usare o **richiamare** una funzione come `lexical_diversity()` digitando il suo nome, seguito da aperte parentesi tonde, il nome del testo, e poi chiuse parentesi. Queste parentesi torneranno spesso; il loro ruolo è separare il nome del compito, come `lexical_diversity()`, dai dati sul quale il compito deve agire, come `text3`. Il valore dei dati che mettiamo tra parentesi quando richiamiamo una funzione è, per questa, l'**argomento**.

Avete già trovato svariate funzioni in questo capitolo, come `len()`, `set()`, e `sorted()`.. Per convenzione aggiungeremo sempre un paio di parentesi vuote dopo il nome di una funzione, come in `len()`, solo per assicurarci che quello di cui stiamo parlando sia una funzione piuttosto che qualche altro tipo di espressione Python. Le funzioni sono un concetto importante nella programmazione, e le menzioniamo all'inizio per dare ai principianti un senso del potere e della creatività della programmazione. Non preoccupatevi se vi sembra un po' confusionario per il momento.

In seguito vedremo come usare le funzioni per tabulare dati, come nella Tabella 1.1. Ogni riga della tabella avrà a che fare con lo stesso calcolo ma dati differenti, e svolgeremo questo lavoro ripetitivo usando una funzione.

Tabella 1.1:

Diversità lessicale dei vari generi nel *Corpus Brown*

Genere	Gettoni	Tipi	Diversità lessicale
Abilità e hobby	82345	11935	6.9
humor	21695	5017	4.3
fiction: scienza	14470	3233	4.5
stampa: reportage	100554	14394	7.0
fiction: romanzo	70022	8452	8.3
religione	39399	6373	6.2

1.2 Un approfondimento su Python: Testi come liste di parole

Avete visto alcuni importanti elementi del linguaggio di programmazione Python. Prendiamoci qualche momento per riesaminarli in maniera sistematica.

Liste

Cos'è un testo? Da un lato è una sequenza di simboli su di una pagina come questa. Da un altro lato è una sequenza di caratteri costituiti da una sequenza di sezioni in cui ogni sezione è una sequenza di paragrafi e così via. Ad ogni modo, per i nostri scopi, considereremo un testo come nient'altro che una sequenza di parole e punteggiatura. Ecco come rappresentiamo un testo in Python, in questo caso la sequenza d'apertura di *Moby Dick*:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>>
```

Dopo il prompt a cui abbiamo dato il nome `sent1` che abbiamo creato, seguito dal simbolo dell'uguale ed alcune parole tra virgolette, separate da virgole, e messe tra parentesi quadre. Questo materiale tra parentesi quadre è conosciuto come **lista** in Python: è così che immagazziniamo un testo. Possiamo ispezionarlo digitandone il nome [1]. Possiamo chiederne la lunghezza [2]. Possiamo persino applicarvi la nostra funzione di diversità lessicale [3].

```
>>> sent1 [1]
['Call', 'me', 'Ishmael', '.']
>>> len(sent1) [2]
4
>>> lexical_diversity(sent1) [3]
1.0
>>>
```

Qualche altra lista è stata predefinita per voi, una per la sequenza d'apertura per ognuno dei nostri testi, `sent2` … `sent9`.. Qui ne passiamo in rassegna due; potete vedere il resto

da soli usando l'interprete Python (se vi si presenta un errore in cui si dice che non è predefinito, dovete prima digitare `from nltk.book import *`).

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

Nota

Il vostro turno: Create voi qualche frase, digitando un nome, il simbolo dell'uguale e una lista di parole come questa: `ex1 = ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']`. Ripetete alcune delle altre operazioni Python che abbiamo visto in precedenza nella Sezione 1.1, es., `sorted(ex1)`, `len(set(ex1))`, `ex1.count('the')`.

Una piacevole sorpresa è che possiamo usare l'operazione d'aggiunta di Python sulle liste. Aggiungendo due liste [1] si crea una nuova lista con tutto quello che c'è sulla prima lista, seguito da tutto quello della seconda lista:

```
>>> ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail'] [1]
['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

Nota

L'uso special dell'operazione di addizione è chiamata **concatenazione**; combina insieme le liste in una singola lista. Possiamo concatenare frasi per costruire un testo.

Non è nemmeno necessario digitare letteralmente le liste; possiamo usare nomi brevi che si riferiscono a liste predefinite.

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

Cosa fare se volessimo aggiungere un singolo oggetto ad una lista? Quest'operazione è conosciuta come **apposizione**. Quando vogliamo apporre ad una lista, la lista si aggiorna da sola come risultato dell'operazione.

```
>>> sent1.append("Some")
>>> sent1
['Call', 'me', 'Ishmael', '.', 'Some']
>>>
```

Indicizzare le liste

Come abbiamo visto, un testo in Python è una lista di parole rappresentate usando una combinazione di parentesi quadre e virgolette. Come avviene con una normale pagina di testo, noi possiamo contare il numero totale delle parole nel `text1` con `len(text1)`, e contare le occorrenze in un testo di una parola specifica, ad esempio `'heaven'` usando `text1.count('heaven')`.

Con un po' di pazienza possiamo selezionare la prima, la 173esima o addirittura la 14,278esima parola di un testo stampato. Analogamente possiamo identificare gli elementi di una lista Python in base al loro ordine di occorrenza nella lista. Il numero che rappresenta questa posizione è un oggetto dell'**indice**. Possiamo istruire Python affinché ci mostri gli oggetti che ricorrono in un indice come in un testo scrivendo il nome di un testo seguito dall'indice tra parentesi quadre.

```
>>> text4[173]
'awaken'
>>>
```

Possiamo fare anche l'inverso; data una parola, trovare l'indice di quando è apparsa la prima volta:

```
>>> text4.index('awaken')
173
>>>
```

Gli indici sono un modo comune per accedere alle parole di un testo, o, più in generale, agli elementi di ogni lista. Python, inoltre, ci permette di accedere alle sottoliste, estraendo pratiche porzioni di linguaggio da ampi testi, una tecnica conosciuta come **affettare**.

```
>>> 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']
>>>
```

Gli indici hanno alcune complessità, indagheremo le seguenti con l'aiuto di una frase artificiale:

```
>>> sent = ['word1', 'word2', 'word3', 'word4', 'word5',
...         'word6', 'word7', 'word8', 'word9', 'word10']
```

```
>>> sent[0]
'word1'
>>> sent[9]
'word10'
>>>
```

Notate come i nostri indici partano da zero: elemento zero, scritto `sent[0]`, è la prima parola, `'word1'`, mentre elemento 9 è `'word10'`. La ragione è semplice: dal momento in cui Python accede al contenuto di una lista dalla memoria del computer, si trova già al primo elemento; dobbiamo dirgli di quanti elementi andare avanti. Di conseguenza, con zero passi avanti rimane al primo elemento.

La pratica di contare da zero può inizialmente confondere, ma è tipica dei moderni linguaggi di programmazione. Ci prenderete velocemente la mano se avete imparato a padroneggiare il sistema di calcolo dei secoli in cui 19XY è un anno del ventesimo secolo o se vivete in un paese in cui i piani di un edificio sono numerati a partire da 1, quindi salire di *n-1* rampe di scale vi porta al livello *n*.

Adesso, se usassimo accidentalmente un indice troppo ampio, avremmo un errore:

```
>>> sent[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Questa volta non è un errore sintattico perché il frammento di programma è sintatticamente corretto. Invece, è un **errore di esecuzione** che produce un messaggio che mostra il contesto dell'errore, seguito dal nome dell'errore, `IndexError`, e una breve spiegazione.

Analizziamo più attentamente lo slicing usando ancora la nostra frase artificiale. Qui verifichiamo che la fetta `5:8` includa elementi agli indici 5, 6 e 7:

```
>>> sent[5:8]
['word6', 'word7', 'word8']
>>> sent[5]
'word6'
>>> sent[6]
'word7'
>>> sent[7]
'word8'
>>>
```

Per convenzione, `m:n` significa elementi *m*…*n-1*. Come dimostra il prossimo esempio, possiamo omettere il primo numero se la fetta comincia all'inizio della lista [1], e possiamo omettere il secondo numero se la fetta arriva alla fine [2]:

```
>>> sent[:3] [1]
['word1', 'word2', 'word3']
>>> text2[141525:] [2]
['among', 'the', 'merits', 'and', 'the', 'happiness', 'of', 'Elinor',
'and', 'Marianne',
',', 'let', 'it', 'not', 'be', 'ranked', 'as', 'the', 'least',
'considerable', ',',
'that', 'though', 'sisters', ',', 'and', 'living', 'almost', 'within',
'sight', 'of',
'each', 'other', ',', 'they', 'could', 'live', 'without', 'disagreement',
'between',
'themselves', ',', 'or', 'producing', 'coolness', 'between', 'their',
'husbands', '.',
'THE', 'END']
>>>
```

Possiamo modificare un elemento di una lista attribuendolo ad uno dei suoi valori di indice. Nel prossimo esempio mettiamo `sent[0]` alla sinistra di un simbolo dell'uguale [1]. Possiamo anche sostituire un intera fetta con del nuovo materiale [2]. Una conseguenza di quest'ultimo cambiamento è che la lista ha solo quattro elementi ed inserire successivamente un valore genera un errore [3].

```
>>> sent[0] = 'First' [1]
>>> sent[9] = 'Last'
>>> len(sent)
10
>>> sent[1:9] = ['Second', 'Third'] [2]
>>> sent
['First', 'Second', 'Third', 'Last']
>>> sent[9] [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Nota

Il vostro turno: Dedicate qualche minuto a definire da soli una frase e modificate parole individuali e gruppi di parole (fette) usando gli stessi metodi visti in precedenza. Mettete alla prova la vostra conoscenza provando gli esercizi sulle liste alla fine del capitolo.

Variabili

Dall'inizio della Sezione 1.1 avete avuto accesso ai testi chiamati `text1`, `text2`, e così via. Ci ha risparmiato un gran lavoro di battitura il poter fare affidamento su un libro di 250,000 parole con un nome breve come in questo caso! In generale, possiamo creare nomi per tutto ciò che ci interessa calcolare. L'abbiamo fatto noi stessi nelle sezioni precedenti, ad esempio definendo una **variabile** `sent1`, come segue:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>>
```

Linee come questa hanno la forma: *variable* = *expression*. Python valuterà l'espressione e salverà il suo risultato alla variabile. Questo processo è chiamato **assegnazione**. Non genera alcun risultato; dovete digitare la variabile su una linea a sé per esaminarne i contenuti. Il simbolo dell'uguale è leggermente ingannevole, in quanto l'informazione si sposta dal lato destro a quello sinistro. Può aiutare pensarlo come una freccia a sinistra. Il nome della variabile può essere qualsiasi cosa vi piaccia, ad esempio `my_sent`, `sentence`, `xyzzy`.. Deve iniziare con una lettere e può includere numeri e trattini. Ecco alcuni esempi di variabili ed assegnazioni:

```
>>> my_sent = ['Bravely', 'bold', 'Sir', 'Robin', ',', 'rode',
... 'forth', 'from', 'Camelot', '.']
>>> noun_phrase = my_sent[1:4]
>>> noun_phrase
['bold', 'Sir', 'Robin']
>>> wOrDs = sorted(noun_phrase)
>>> wOrDs
['Robin', 'Sir', 'bold']
>>>
```

. Ricordate che le parole in maiuscolo compaiono prima delle parole in minuscolo nelle liste smistate.

Nota

Notate nell'esempio precedente che abbiamo diviso la definizione di `my_sent` tra due linee. Le espressioni Python possono essere divise su linee multiple, a patto che questo accada all'interno di qualsiasi tipo di parentesi. Python utilizza il prompt "`...`" per indicare che è in attesa di ulteriori risultati. Non importa quanta rientranza venga usata in queste linee di prolungamento, ma di solito un po' di rientranza le rende più facili da leggere.

È bene scegliere nomi di variabili significativi per ricordarvi, ed aiutare chiunque altro legga il vostro codice Python, a cosa serve il vostro codice. Python non prova a dare un senso ai nomi; segue ciecamente le vostre istruzioni, e non obietta se fate qualcosa che può creare confusione, come `one = 'two'` o `two = 3`. La sola restrizione è che il nome di una variabile non può essere nessuna delle parola riservate di Python, come `def`, `if`, `not`, e `import`.. Se usate una parola riservata, Python produrrò un errore sintattico:

```
>>> not = 'Camelot'
File "<stdin>", line 1
    not = 'Camelot'
        ^
SyntaxError: invalid syntax
>>>
```

Useremo spesso variabili per mantenere i passi intermedi di un calcolo, specialmente quando rende il codice più facile da seguire. Nonostante `len(set(text1))` possa anche essere scritto:

```
>>> vocab = set(text1)
>>> vocab_size = len(vocab)
>>> vocab_size
19317
>>>
```

Attenzione!

Abbiate cura della scelta dei nomi (o **identificatori**) per le variabili Python. Innanzitutto, dovreste iniziare il nome con una lettera, opzionalmente seguita da una cifra (da `0` a `9`) o lettere. Quindi `abc23` va bene, ma causerà un errore di sintassi. I nomi sono sensibili alla maiuscole, per cui significa che `myVar` e `myvar` sono variabili distinte. I nomi delle variabili non posso contenere spazi bianchi, ma potete separare le parole usando un trattino basso, ad esempio `my_var`.. Fate attenzione a non inserire una lineetta al posto di un trattino basso: è sbagliato, in quanto Python lo interpreta come un simbolo del meno.

Stringhe

Alcuni dei metodi che abbiamo usato per accedere agli elementi di una lista funzionano anche con parole individuali, o **stringhe**. Ad esempio, possiamo assegnare una stringa ad una variabile [1], indice ad una stringa [2], e una fetta ad una stringa [3]:

```
>>> name = 'Monty' [1]
>>> name[0] [2]
'M'
>>> name[:4] [3]
'Mont'
>>>
```

Possiamo anche effettuare moltiplicazione ed addizione con le stringhe:

```
>>> name * 2
'MontyMonty'
>>> name + '!'
'Monty!'
>>>
```

Possiamo unire le parole di una lista per formare una singola stringa, o dividere una stringa in una lista, come segue:

```
>>> ' '.join(['Monty', 'Python'])
'Monty Python'
>>> 'Monty Python'.split()
['Monty', 'Python']
>>>
```

Ritorneremo sull'argomento delle stringhe nel Capitolo 3. Per adesso, abbiamo due importanti blocchi in costruzione, liste e stringhe, e sono pronti per tornare ad applicarsi all'analisi del linguaggio.

1.3 Calcolare con la lingua: Semplici statistiche

Torniamo alla nostra esplorazione dei modi in cui possiamo portare le nostre risorse di calcolo a sostenere ampie quantità di testo. Abbiamo iniziato questa discussione nella Sezione 1.1, ed abbiamo visto come ricercare le parole nel contesto, come compilare il vocabolario di un testo, come generare un testo casuale nello stesso stile, e così via.

In questa sezione affrontiamo la domanda di cosa rende un testo distinto, ed usare metodi automatici per trovare parole caratteristiche ed espressioni di un testo. Come nella Sezione 1.1 potete trovare nuove caratteristiche del linguaggio Python copiandole nell'interprete ed apprenderete queste caratteristiche sistematicamente nella sezione seguente.

Prima di procedere, potreste voler mettere alla prova la vostra conoscenza dell'ultima sezione prevedendo il risultato del seguente codice. Potete usare l'interprete per controllare se avete fatto bene. Se non siete sicuri di come eseguire questo compito, sarebbe opportuno rivedere la sezione precedente prima di procedere.

```
>>> saying = ['After', 'all', 'is', 'said', 'and', 'done',
...           'more', 'is', 'said', 'than', 'done']
>>> tokens = set(saying)
>>> tokens = sorted(tokens)
>>> tokens[-2:]
what output do you expect here?
>>>
```

Distribuzioni di frequenza

Come possiamo identificare automaticamente la parole di un testo che danno più informazioni sull'argomento e il genere del testo? Immaginate cosa fare per trovare le cinquanta parole più frequenti in un libro. Un metodo potrebbe essere tenere il conto di ogni vocabolo, come quello mostrato nella Figura 1.3. Il conteggio avrebbe bisogno di migliaia di righe che richiederebbero un processo eccessivamente laborioso, talmente laborioso da assegnare il compito ad una macchina.

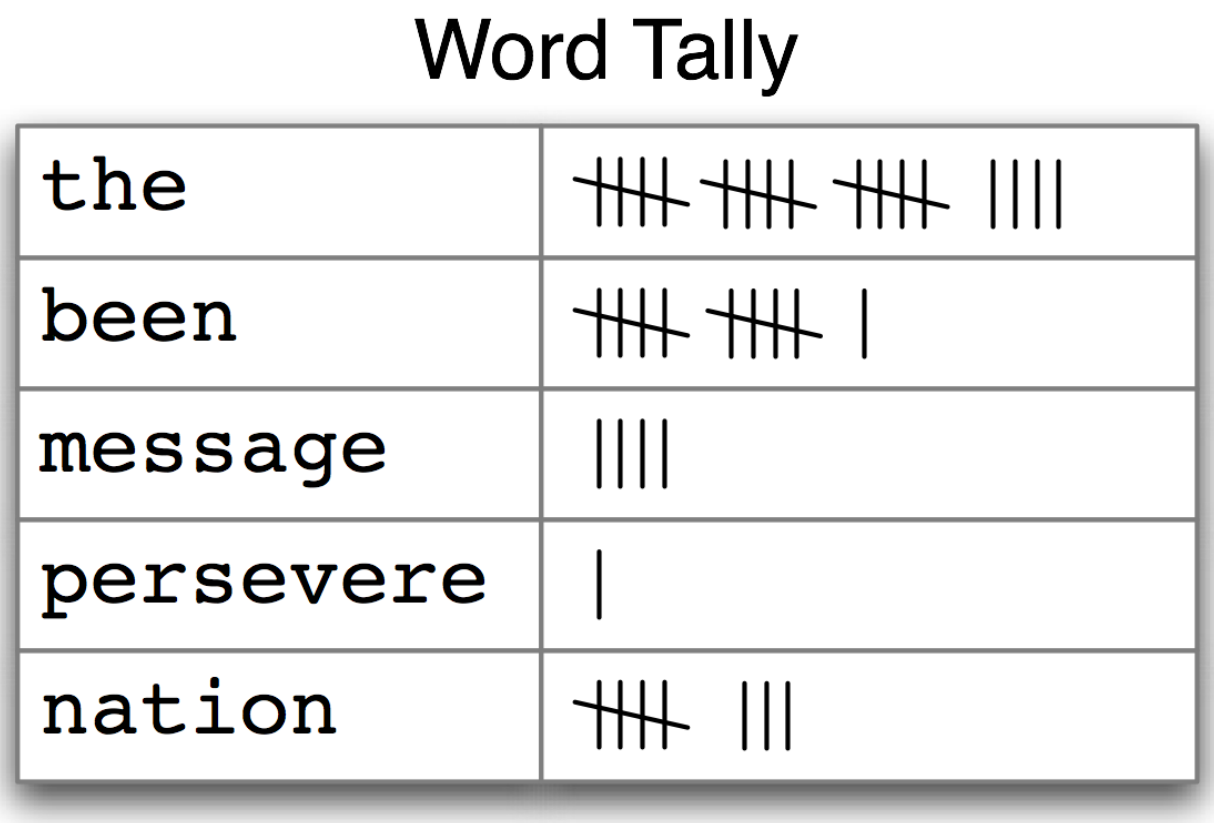


Figura 1.3: Conteggio della parole presenti in un testo (distribuzione di frequenza)

La tabella nella Figura 1.3 è conosciuta come una **distribuzione di frequenza** e ci dice la frequenza di ogni vocabolo nel testo. (In generale, potrebbe contare ogni genere di evento riscontrabile.) Si tratta di una "distribuzione" perché ci dice come il numero totale di gettoni di parole nel testo siano distribuiti tra i vocaboli. In quanto abbiamo spesso bisogno di distribuzioni di frequenza nel linguaggio di programmazione, NLTK ci mette a disposizione un supporto integrato. Usiamo `FreqDist` per trovare le 50 parole più frequenti in *Moby Dick*. Cercate di decifrare cosa sta succedendo, dopodiché leggete la spiegazione che segue.

```
>>> fdist1 = FreqDist(text1) [1]
>>> fdist1 [2]
<FreqDist with 260819 outcomes>

>>> vocabulary1 = fdist1.keys() [3]

>>> vocabulary1[:50]
[',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in', 'that', "'", '-',
'his', 'it', 'I', 's', 'is', 'he', 'with', 'was', 'as', '"', 'all',
'for',
'this', '!', 'at', 'by', 'but', 'not', '--', 'him', 'from', 'be', 'on',
'so', 'whale', 'one', 'you', 'had', 'have', 'there', 'But', 'or', 'were',
'now', 'which', '?', 'me', 'like']
>>> fdist1['whale']
906
>>>
```

Non appena invochiamo `FreqDist`, passiamo il nome del testo come argomento [1]. Possiamo indagare il numero totale di parole ("esiti") che sono state conteggiate [2], 260,819 nel caso di *Moby Dick*. L'espressione `keys()` ci dà una lista di tutti i distinti generi nel testo [3], e possiamo guardare ai primi 50 affettando la lista [4].

Nota

Il vostro turno: Provate da soli sul `text2`. l'esempio precedente di distribuzione di frequenza. Fate attenzione ad usare correttamente le parentesi e le lettere maiuscolo. Se ottenete un messaggio d'errore `NameError: name 'FreqDist' is notdefined`, dovete iniziare il vostro lavoro con `from nltk.book import *`

Qualche parola prodotta nell'ultimo esempio ci aiuta a carpire l'argomento o il genere di questo testo? Solo una, *balena*, è minimamente esplicativa! Ritorna più di 900 volte. Il resto delle parole non ci dice nulla sul testo; sono solo "connettivi" della lingua inglese. Quale proporzione del testo è occupata da queste parole? Possiamo generare un grafico di frequenza cumulativo per queste parole, usando `fdist1.plot(50, cumulative=True)`, per produrre il diagramma nella Figura 1.4. Queste 50 parole ammontano a quasi la metà del libro!

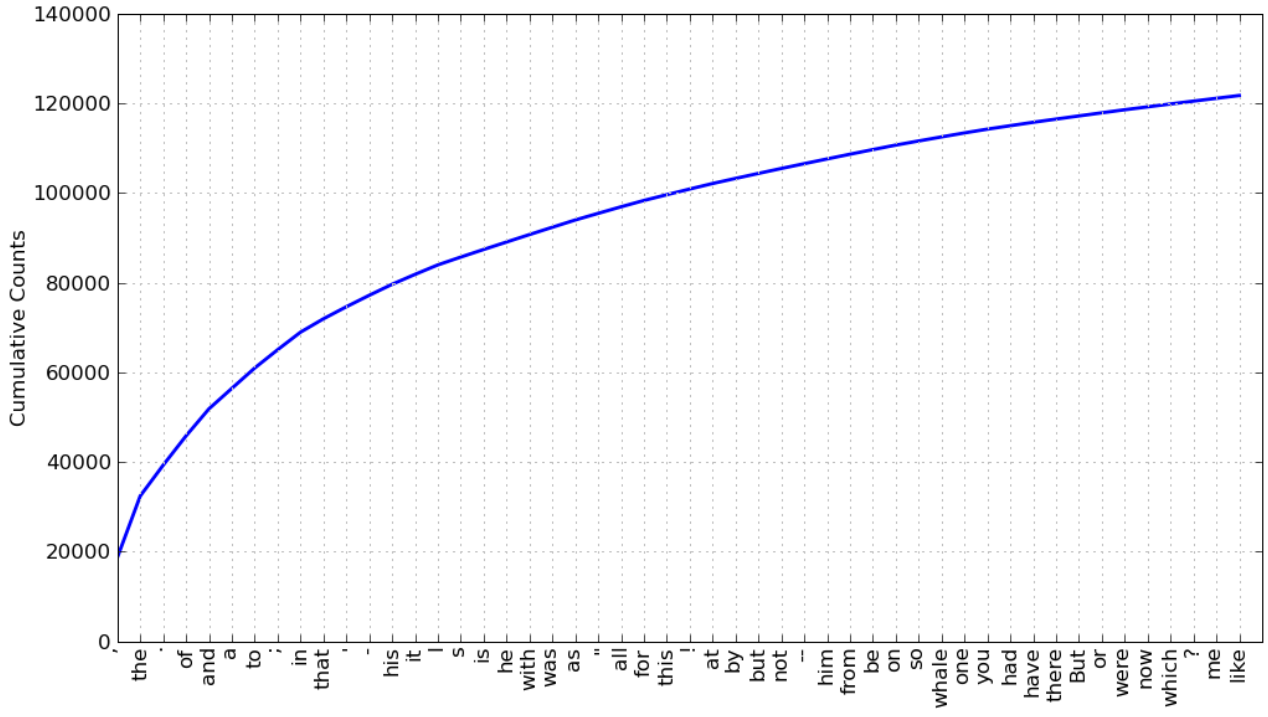


Figura 1.4: Grafico di frequenza cumulativa per le 50 parole più frequenti in Moby Dick: ammontano a circa la metà dei gettoni.

Se le parole frequenti non ci aiutano, che dire delle parole che ricorrono solo una volta, le cosiddette **hapax**? Visionale digitando `fdist1.hapaxes()`. Questa lista contiene *lexicographer*, *cetological*, *contraband*, *expostulations*, e circa 9,000 altre. Sembra che ci

siano troppo parole rare, e senza vedere il contesto probabilmente non capiremmo cosa vogliono dire metà degli hapax in ogni caso! In quanto né le parole frequenti che quelle infrequenti ci aiutano, bisogna escogitare qualcos'altro.

Selezione a grana fine di parole

Continuiamo, diamo uno sguardo alle parole *lunghe* di un testo; forse queste saranno più caratteristiche ed esplicative. Per farlo adattiamo qualche notazione dalla serie teorica. Vorremmo trovare le parole dal vocabolario del testo che sono più lunghe di 15 caratteri. Chiamiamo questo proprietà *P*, così che *P(w)* sia vero se e solo se w è più lungo di 15 caratteri. Adesso possiamo esprimere le parole interessate usando una serie matematica di notazione come mostrato in (1a). Questo significa che "la serie di tutte le *w* tale che *w* è un elemento di *V* (il vocabolario) e w hanno la proprietà *P*".

(1) a. $\{w \mid w \in V \ \& \ P(w)\}$
b. `[w for w in V if p(w)]`

L'espressione Python corrispondente è data in (1b). (Notate che produce una lista, non una serie, il che significa che c'è la possibilità di duplicati.) Osservate quanto siano simili le due notazioni. Avanziamo di un passo e scriviamo il codice Python eseguibile:

```
>>> 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']
>>>
```

Per ogni parola nel vocabolario `V`, controlliamo se `len(w)` è più grande di 15; tutte le altre parole verranno ignorate. Discuteremo questa sintassi più attentamente dopo.

Nota

Il vostro turno: Provate le affermazioni precedenti nell'interprete Python e sperimentate cambiando il testo e cambiando la lunghezza della condizione. C'è qualche differenza di risultato se cambiate i nomi delle variabili, ad esempio, usando `[word for word in vocab if ...]`?

Torniamo al nostro compito di trovare parole che caratterizzano un test. Notate che le parole lunghe nel riflettono il suo fulcro nazionale, *constituzionalmente*,

transcontinentale, mentre quelle nel `text5` riflettono il suo contenuto informale: *booooooooooglyyyyyy* e *yuuuuuuuuuuuummmmmmmmmmm*. Siamo riusciti ad estrarre automaticamente parole che tipizzano un testo? Bene, queste parole lunghissime sono spesso hapax (ad esempio, unico) e forse sarebbe meglio trovare parole lunghe che *ricorrono con frequenza*. Sembra allettante in quanto elimina le parole corte frequenti (ad esempio *the*) e parole lunghe infrequenti (ad esempio *antiphilosophists*). Ecco tutte le parole del corpus colloquiale che sono più lunghe di sette caratteri e ricorrono 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']
>>>
```

Notate come abbiamo usato due condizioni: `len(w) > 7` assicura che le parole siano più lunghe di sette lettere, e `fdist5[w] > 7` assicura che queste parole ricorrono più di sette volte. Infine siamo riusciti ad identificare automaticamente le parole frequenti portatrici di contenuto ricorrente del testo. È un traguardo modesto ma importante: una piccola parte di codice che processa decine di migliaia di parole produce qualche risultato informativo.

Collocazioni e bigrammi

Una **collocazione** è una sequenza di parole che ritornano insieme insolitamente spesso. Quindi *vino rosso* è una collocazione, mentre *il vino* no. Una caratteristica delle collocazioni è che non sono impermeabili alla sostituzione con parole che hanno significati simili; ad esempio *vino bordeaux* suono decisamente strano.

Per far pratica con le collocazioni, cominciamo estraendo da un testo una lista di coppia di parole, anche conosciute come **bigrammi**. Ciò si ottiene facilmente con la funzione `bigrams()`:

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

Qui vediamo che la coppia di parole *than-done* è un bigramma, e lo scriviamo in Python come `('than', 'done')`. Adesso, le collocazioni sono essenzialmente solo bigrammi frequenti, eccetto che vogliamo prestare più attenzione ai casi che coinvolgono parole rare. In particolare, vogliamo trovare bigrammi che ricorrono più spesso di quanto ci aspetteremmo sulla base della frequenza di parole individuali. La funzione `collocations()` fa questo per noi (vedremo come funziona più tardi):

```
>>> text4.collocations()
Building collocations list
United States; fellow citizens; years ago; Federal Government; General
Government; American people; Vice President; Almighty God; Fellow
citizens; Chief Magistrate; Chief Justice; God bless; Indian tribes;
public debt; foreign nations; political parties; State governments;
National Government; United Nations; public money
>>> text8.collocations()
Building collocations list
medium build; social drinker; quiet nights; long term; age open;
financially secure; fun times; similar interests; Age open; poss
rship; single mum; permanent relationship; slim build; seeks lady;
Late 30s; Photo pls; Vibrant personality; European background; ASIAN
LADY; country drives
>>>
```

Le collocazioni che emergono sono molto aderenti al genere dei testi. Per trovare *vino rosso* come collocazione, avremmo bisogno di lavorare su una porzione molto più ampia di testo.

Contare altre cose

Il conteggio delle parole è utile, ma possiamo contare anche altre cose. Ad esempio, possiamo guardare alla distribuzione delle lunghezze delle parole in un testo, creando `FreqDist` da una lunga lista di numeri, in cui ogni numero è la lunghezza di una 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])

>>> fdist
<FreqDist with 260819 outcomes>
>>> fdist.keys()
[3, 1, 4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20]
>>>
```

Iniziamo derivando una lista di lunghezze di parole nel `text1` [1], e il `FreqDist` poi conta il numero di volte che ognuna ritorna [2]. Il risultato [3] è una distribuzione contenente un quarto di milione di oggetti, ognuno dei quali è un numero corrispondente ad una parola gettone nel testo. Ma ci sono solo 20 oggetti distinti ad essere contati, i numeri da 1 a 20, perché ci sono solo 20 lunghezze di parole differenti. Ovvero, ci sono parole che consistono in un solo carattere, due caratteri, ..., venti caratteri, ma nessuno con ventuno o più caratteri. Ci si potrebbe domandare quanto siano frequenti le lunghezze differenti di una parola (ad esempio, quante parole di lunghezza quattro appaiono nel testo, ci sono più parole di lunghezza cinque che lunghezza quattro, ecc). Possiamo fare come segue:

```
>>> fdist.items()
[(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)
0.19255882431878046
>>>
```

Da ciò vediamo che la lunghezza di parole più frequente è 3. E che parole di lunghezza 3 ammontano approssimativamente a 50,000 (o 20%) delle parole che costituiscono il libro. Nonostante non lo approfondiremo qui, un'ulteriore analisi della lunghezza delle parole potrebbe aiutarci a capire le differenze tra autori, generi e lingue.

La Tabella 1.2 riassume le funzioni definite nelle distribuzioni di frequenza.

Tabella 1.2:

Funzioni definite per le distribuzioni di frequenza di NLTK

Esempio	Descrizione
`fdist = FreqDist(samples)`	crea una distribuzione di frequenza contenente i campioni dati
`fdist.inc(sample)`	incrementa il conto per questo campione
`fdist['monstrous']`	conto del numero di volte che ricorrono nel dato campione
`fdist.freq('monstrous')`	frequenza di un dato campione
`fdist.N()`	numero totale dei campioni
`fdist.keys()`	i campioni smistati per ordine di frequenza decrescente
`for sample in fdist:`	ripetere i campioni, in ordine di frequenza decrescente
`fdist.max()`	campione con il conto più grande
`fdist.tabulate()`	tabulare la distribuzione di frequenza
`fdist.plot()`	grafico della frequenza di distribuzione
`fdist.plot(cumulative=True)`	grafico cumulativo della frequenza di distribuzione
`fdist1 < fdist2`	prova se i campioni in `fdist1` ricorrono meno frequentemente che in `fdist2`

La nostra trattazione delle distribuzioni di frequenza ha introdotto alcuni importanti concetti di Python, e li analizzeremo sistematicamente nella Sezione 1.4.

1.4 Ritorno a Python: Prendere decisioni ed avere il controllo

Finora i nostri piccoli programmi hanno avuto alcune qualità interessanti: la capacità di lavorare con il linguaggio, ed il potenziale di risparmiare sforzo umano attraverso l'automazione. Un tratto fondamentale della programmazione è la capacità delle macchine di prendere decisioni per nostro conto, eseguendo istruzioni quando si incorre in talune condizioni, o riciclando ripetutamente i dati del testo fin quando alcune condizioni non vengono soddisfatte. Quest'aspetto è conosciuto come **controllo**, ed è il fulcro di questa sezione.

Condizionali

Python supporta una vasta gamma di operatori, come < and >=, mettere alla prova la relazione tra i valori. La serie completa di questi **operatori relazionali** è mostrata nella Tabella 1.3.

Tabella 1.3:

Operatori di comparazione numerica

Operatore	Relazione
<	Minore di
<=	minore o uguale a
==	uguale ad (notate che si tratta di due simboli, non uno
!=	non uguale a
>	maggiore di
>=	maggiore di o uguale a

Possiamo usarli per selezionare parole differenti da una frase di testo informativo. Ecco alcuni esempi, solo l'operatore viene cambiato da una linea a quella dopo. Usano tutte `sent7`, la prima frase del `text7` (*Wall Street Journal*). Come prima, se ottenete un errore che dice che `sent7` è indefinito, c'è bisogno di digitare prima: `from nltk.book import *`

C'è un modello comune per tutti questi esempi: `[w for w in text if` *condition* `]`, in cui è un "test" Python che produce sia il vero che il falso. Nei casi mostrati nell'esempio del codice precedente, la condizione è sempre una comparazione numerica. Comunque, possiamo anche mettere alla prova varie proprietà delle parole usando le funzioni elencate nella Tabella 1.4.

Tabella 1.4:

Alcuni operatori di comparazione di parole

Funzione	Significato
`s.startswith(t)`	controlla che s cominci con t
`s.endswith(t)`	controlla che s finisca con t
`t in s`	controlla che t contenga al suo interno s
`s.islower()`	controlla se tutti i caratteri del caso in s sono minuscoli
`s.isupper()`	controlla se tutti i caratteri del caso in s sono maiuscoli
`s.isalpha()`	controlla se tutti i caratteri in s sono alfabetici
`s.isalnum()`	controlla se tutti i caratteri in s sono alfanumerici
`s.isdigit()`	controlla se tutti i caratteri in s sono cifre
`s.istitle()`	controlla se s ha le lettere del titolo in maiuscolo

Ecco alcuni esempi di questi operatori in uso per selezionare parole dai nostri testi: parole che finiscono per –ableness; parole contenenti gnt; parole aventi un iniziale maiuscola; e parole consistenti interamente di 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']
>>>
```

Possiamo anche creare condizioni più complesse. Se c è una condizione, poi `not` c è anch'essa una condizione. Se abbiamo due condizione c_1 e c_2, possiamo combinarle per formare una nuova condizione usando la congiunzione e la disgiunzione: c_1 `e` c_2, c_1 `o` c_2.

Nota

Il vostro turno: Scorrete i seguenti esempi e provate a spiegare cosa succede in ognuno. Poi provate a creare alcune condizioni da soli.

```
>>> sorted([w for w in set(text7) if '-' in w and 'index' in w])
>>> sorted([wd for wd in set(text3) if wd.istitle() and len(wd) > 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])
```

Operare su ogni elemento

Nella sezione 1.3, abbiamo visto alcuni esempi per calcolare oggetti che non sono parole. Diamo uno sguardo più ravvicinato alla notazione che abbiamo usato:

```
>>> [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, ...]
>>> [w.upper() for w in text1]
['[', 'MOBY', 'DICK', 'BY', 'HERMAN', 'MELVILLE', '1851', ']',
'ETYMOLOGY', '.', ...]
>>>
```

Queste espressioni hanno la forma `[f(w) for ...]` or `[w.f() for ...]`, in cui è una funzione che opera su una parola per calcolarne la lunghezza, o convertirla in maiuscolo. Per adesso, non avete bisogno di capire la differenza tra notazioni `f(w)` e `w.f()`. Invece, apprendete semplicemente questo idioma Python che svolge la stessa operazione su ogni elemento di una lista. Negli esempi precedenti, passa in rassegna ogni parola nel `text1`, assegnando ad ognuna a turno la variabile e svolgendo l'operazione specifica sulla variabile.

Nota

La notazione appena descritta è chiamata "lista di comprensione." Questo è il nostro primo esempio dell'idioma Python, una notazione fissa che usiamo abitualmente senza preoccuparci di analizzarla ogni volta. Imparare a controllare questi idiomi è una parte importante per diventare un programmatore fluente in Python.

Torniamo alla questione della taglia del vocabolario ed applichiamo lo stesso idioma qui:

```
>>> len(text1)
260819
>>> len(set(text1))
19317
>>> len(set([word.lower() for word in text1]))
17231
>>>
```

Ora che non contiamo due volte parole come Questo e questo, che differiscono solo nelle maiuscole, ne abbiamo cancellate 2,000 dal conto del vocabolario! Possiamo avanzare di un passo ed eliminare i numeri e la punteggiatura dal conto del vocabolario estromettendo ogni oggetto non alfabetico:

```
>>> len(set([word.lower() for word in text1 if word.isalpha()]))
16948
>>>
```

Quest'esempio è leggermente complicato: converte in minuscolo tutti gli oggetti puramente alfabetici. Forse sarebbe stato più semplice contare solo gli oggetti minuscoli, ma questo ci dà la risposta sbagliata (perché?).

Non preoccupatevi se non vi sentite ancora sicuri con le comprensioni delle liste, in quanto vedrete molti più esempi e spiegazioni nei capitoli a seguire.

Blocchi di codice annidati

Molti linguaggi di programmazione ci permettono di eseguire una porzione di codice quando un'**espressione condizionale** o l'asserzione `if` viene soddisfatta. Abbiamo già visto esempi di testi condizionali in codice come `[w for w in sent7 if len(w) < 4]`. Nel programma seguente, abbiamo creato una variabile chiamata `word` contenente il valore di stringa `'cat'`. L'asserzione `if` controlla se il test `len(word) < 5`è vero. Lo è, quindi il corpo dell'asserzione `if` viene invocato e l'asserzione `print` viene eseguita, visualizzando un messaggio all'utente. Ricordate di mandare a capo l'asserzione `print` digitando quattro spazi.

```
>>> word = 'cat'
>>> if len(word) < 5:
...     print 'word length is less than 5'
...   [1]
word length is less than 5
>>>
```

Quando usiamo l'interprete Python dobbiamo aggiungere una linea vuota [1] in più affinché avverta che il blocco annidato è completo.

Se cambiamo il test condizionale a `len(word) >= 5` , per controllare che la lunghezza di `word` è superiore o uguale a 5, allora il test non sarà più vero. Questa volta, il corpo dell'asserzione `if` non verrà eseguito e nessun messaggio verrà mostrato all'utente:

```
>>> if len(word) >= 5:
...   print 'word length is greater than or equal to 5'
...
>>>
```

Un'asserzione `if` è conosciuta come **struttura di controllo** perché controlla se il codice nella porzione a capo funzionerà. Un'altra struttura di controllo è il ciclo `for` . Provate ciò che segue e ricordate di includere i due punti ed i quattro spazi:

```
>>> for word in ['Call', 'me', 'Ishmael', '.']:
...     print word
...
Call
me
Ishmael
.
>>>
```

Questo viene chiamato ciclo perché Python esegue il codice in maniera circolare. Comincia svolgendo l'asserzione `word = 'Call'`, usando effettivamente la variabile `word` per nominare il primo oggetto della lista. Poi, visualizza il valore di `word` all'utente. In seguito, ritorna all'asserzione `for`, e svolge l'asserzione `word = 'me'`, prima di visualizzare questo nuovo valore all'utente, e così via. Continua in questa maniera finché ogni oggetto della lista viene processato.

Circolare con condizioni

Ora possiamo combinare le asserzioni `if` e `for` . Faremo circolare ogni oggetto della lista, e pubblicheremo l'oggetto solo se termina con la lettera *l*. sceglieremo un altro nome per la variabile per dimostrare che Python non cerca di dare un senso ai nomi delle variabili.

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>> for xyzzy in sent1:
...     if xyzzy.endswith('l'):
...         print xyzzy
...
Call
Ishmael
>>>
```

Noterete che le asserzioni `if` e `for` hanno due punti alla fine della riga, prima del capoverso. Infatti, tutte le strutture di controllo Python finiscono con due punti. I due punti indicano che l'asserzione attuale si riferisce al seguente blocco a capo.

Possiamo anche specificare un'azione da eseguire se la condizione dell'asserzione `if` non viene raggiunta. Qui vediamo l'asserzione `elif` (altro se), e l'asserzione `else` . Notate che anche queste hanno due punti prima del codice a capo.

```
>>> for token in sent1:
...     if token.islower():
...         print token, 'is a lowercase word'
...     elif token.istitle():
...         print token, 'is a titlecase word'
...     else:
...         print token, 'is punctuation'
```

```
...
Call is a titlecase word
me is a lowercase word
Ishmael is a titlecase word
. is punctuation
>>>
```

Come potete vedere, anche con poca conoscenza di Python, potete cominciare a costruire programmi Python multilinee. È importante sviluppare tali programmi in parti, controllando che ogni parte faccia ciò che volete prima di combinarle insieme in un programma. Ecco perché l'interprete Python interattivo è inestimabile, e perché dovreste imparare ad usarlo agilmente.

Finalmente, combiniamo gli idiomi che abbiamo esplorato. Innanzitutto creiamo una lista di parole *cie* e *cei*, poi applichiamo il loop su ogni oggetto e lo pubblichiamo. Notate la virgola alla fine dell'asserzione pubblicata che dice a Python di produrre il suo risultato su una singola linea.

```
>>> tricky = sorted([w for w in set(text2) if 'cie' in w or 'cei' in w])
>>> for word in tricky:
...     print word,
ancient ceiling conceit conceited conceive conscience
conscientious conscientiously deceitful deceive ...
>>>
```

1.5 Comprendere il linguaggio naturale automatico

Siamo esplorando il linguaggio dalla base, con l'aiuto di testi e del linguaggio di programmazione Python. Ad ogni modo, siamo anche interessati a sfruttare la nostra conoscenza del linguaggio e del calcolo per costruire tecnologie di linguaggio utili. Ora coglieremo l'opportunità di allontanarci dal nocciolo del codice per dipingere un'immagina più grande del processo di linguaggio naturale.

Ad un livello puramente pratico, abbiamo tutti bisogno di navigare l'universo dell'informazione rinchiuso nel testo sulla rete. I motori di ricerca sono cruciali per la crescita e la popolarità del web, ma hanno alcuni punti deboli. Ci vogliono abilità, conoscenza ed un po' di fortuna per ricavare risposte da domande come: *Quali siti turistici posso visitare tra Philadelphia e Pittsburgh con un budget limitato? Cosa pensano gli esperti delle fotocamere digitali SLR? Quali pronostici a proposito del mercato dell'acciaio furono fatti da commentatori credibili nella settimana passata?* Avere un computer per rispondere automaticamente coinvolge una serie di mansioni di programmazione del linguaggio, incluse l'estrazione delle informazioni, la deduzione ed il riassunto, ed avrebbe bisogno di essere portata su una bilancia ad un livello di robustezza che è oltre le nostre attuali capacità.

Ad un livello più filosofico, una sfida a lungo termine con l'intelligenza artificiale è stata costruire macchine intelligenti, ed una parte cruciale del comportamento intelligente è capirne il linguaggio. Per molti anni l'obiettivo è stato visto molto difficilmente. Ad ogni

modo, come le tecnologie NLP sono maturate, e metodi robusti per analizzare testi illimitati sono diventati più diffusi, la prospettiva della comprensione del linguaggio naturale è riemersa come obiettivo plausibile

In questa sezione descriviamo alcune tecnologie di comprensione del linguaggio, per darvi un'idea delle sfida interessanti che vi attendono.

Disambiguazione del senso delle parole

Nella **disambiguazione del senso delle parole** vogliamo capire quale senso di una parola si intenda in un dato contesto. Prendete in considerazione le parole ambigue *servizio e piatto*:

(2) a. *servizio*: aiuto con cibo o bevande; avere un incarico; mettere la palla in gioco
b. *piatto*: portata di un pasto, dispositivo di comunicazione

In una frase contenente l'espressione: servì il piatto, potete rilevare che sia *servizio* che *piatto* vengono usati con i loro significati di cibo. È inusuale che l'argomento di discussione passi dallo sport alle stoviglie nel giro di tre parole. Questo vi costringerebbe ad inventare immagini bizzarre, come un tennista che sfoga la frustrazione su un servizio da the in porcellana steso accanto al campo. In altre parole, disambiguiamo automaticamente le parole usando il contesto, sfruttando il semplice fatto che parole attigue hanno significati collegati. Come altro esempio di questo effetto contestuale, considerate la parola by che ha svariati significati, ad esempio: *the book by Chesterton* (d'agente, Chesterton era l'autore del libro); *the cup by the stove* (locativo, il fornello è dove si trova la tazza); e *submit by Friday* (temporale, Venerdì è il giorno della consegna). Osservate in (3c) che il significato della parola in corsivo ci aiuta ad interpretare il significato di *by*.

(3) a. The lost children were found by the *searchers* (d'agente)
b. The lost children were found by the *mountain* (locativo)
c. The lost children were found by the *afternoon* (temporale)

La risoluzione del pronome

Un tipo di comprensione più profonda del linguaggio è trovare "chi ha fatto cosa a chi", ovvero, indagare i soggetti e gli oggetti dei verbi. Avete imparato a farlo alle scuole elementari ma è più difficile di quanto potreste pensare. Nella frase *i ladri hanno rubato i dipinti* è facile dire chi ha compiuto l'azione di rubare. Considerate le seguenti tre possibili seguenti frasi in (4c), e provate a determinare cosa è stato venduto, catturato e trovato (un caso è ambiguo).

(4) a. I ladri hanno rubato i dipinti. Sono stati successivamente *venduti*.
b. I ladri hanno rubato i dipinti. Sono stati successivamente *catturati*.
c. I ladri hanno rubato I dipinti. Sono stati successivamente *trovati*.

Rispondere a questa domanda significa trovare l'**antecedente** del pronome *essi*, sia ladri che dipinti. Le tecniche di calcolo per affrontare questo problema comprendono la

risoluzione d'anafora, ovvero identificare a cosa si riferisce un pronome o il nome di una frase, e la **categorizzazione del ruolo semantico**, ovvero identificare come un nome della frase si relaziona al verbo (come agente, paziente, strumento e così via).

Generare un prodotto di linguaggio

Se possiamo risolvere automaticamente tali problemi di comprensione della lingua, saremo in grado di passare a compiti che coinvolgono la generazione di prodotti di linguaggio, come **rispondere alle domande** e la **traduzione meccanica**. Nel primo caso, una macchina dovrebbe essere in grado di rispondere alle domande di un utente riguardanti una raccolta di testi:

(5) a. *Testo:* ... I ladri hanno rubato i dipinti. Sono stati successivamente venduti ...
b. *Umano:* Chi o cosa è stato venduto?
c. *Macchina:* I dipinti.

La risposta della macchina dimostra che ha decifrato correttamente che si riferivano ai dipinti e non ai ladri. Nel secondo caso, la macchina dovrebbe essere in grado di tradurre il testo in un'altra lingua, esprimendo accuratamente il significato del testo originale. Traducendo il testo dell'esempio in francese, siamo obbligati a scegliere il genere del pronome nella seconda frase: *ils* (maschile) se i ladri sono stati venduti, ed *elles* (femminile) se i dipinti sono stati venduti. La corretta traduzione dipende davvero dalla corretta comprensione del pronome.

(6) a. I ladri hanno rubato i dipinti. Sono stati successivamente trovati.
b. Les voleurs ont volé les peintures. Ils ont été trouvés plus tard. (i ladri)
c. Les voleurs ont volé les peintures. Elles ont été trouvées plus tard. (i dipinti)

In tutti questi esempi, decifrare il senso di una parola, il soggetto di un verbo, e l'antecedente di un pronome sono passi per stabilire il significato di una frase, cose che ci aspetteremmo che un sistema di comprensione della lingua sia in grado di fare.

Traduzione meccanica

Per un lungo periodo, la traduzione meccanica (MT) è stata il Sacro Graal della comprensione linguistica, cercando in definitiva di fornire una traduzione idiomatica di alta qualità tra ogni coppia di lingue. Le sue radici affondano ai primi giorni della Guerra Fredda quando la promessa di una traduzione automatica condusse al sostanziale finanziamento del governo, e con esso, alla genesi del NLP stesso.

Oggi, sistemi di traduzione pratica esistono per particolari coppie di lingue, ed alcune sono integrate in motori di ricerca del web. Ad ogni modo, questi sistemi hanno alcuni seri punti deboli. Possiamo esaminarli con l'aiuto del "babelizer" di NLTK (che viene caricato automaticamente quando si importa il materiale di questo capitolo usando `fromnltk.book import *`). Questo programma converte la frase da tradurre in un linguaggio specifico, poi riconverte in inglese la frase risultante da tradurre. Si ferma

dopo 12 ripetizioni, o se riceve una traduzione che sia stata già prodotta (indicando un ciclo):

```
>>> babelize_shell()
NLTK Babelizer: type 'help' for a list of commands.
Babel> how long before the next flight to Alice Springs?
Babel> german
Babel> run
0> how long before the next flight to Alice Springs?
1> wie lang vor dem folgenden Flug zu Alice Springs?
2> how long before the following flight to Alice jump?
3> wie lang vor dem folgenden Flug zu Alice springen Sie?
4> how long before the following flight to Alice do you jump?
5> wie lang, bevor der folgende Flug zu Alice tun, Sie springen?
6> how long, before the following flight to Alice does, do you jump?
7> wie lang bevor der folgende Flug zu Alice tut, tun Sie springen?
8> how long before the following flight to Alice does, do you jump?
9> wie lang, bevor der folgende Flug zu Alice tut, tun Sie springen?
10> how long, before the following flight does to Alice, do do you jump?
11> wie lang bevor der folgende Flug zu Alice tut, Sie tun Sprung?
12> how long before the following flight does leap to Alice, does you?
```

Osservate che il sistema traduce correttamente *Alice Springs* dall'inglese al tedesco (nella linea che inizia per `1>`), ma ritraducendo in inglese finisce per diventare *Alice jump* (linea 2). La preposizione *before* è inizialmente tradotta nella corrispondente preposizione tedesca vor, ma dopo nella congiunzione *bevor* (linea 5). Dopo la linea 5 le frasi sono diventate senza senso (ma notate la varia formulazione indicata dalle virgole, e il cambio da *jump* a *leap*). Il sistema di traduzione non ha riconosciuto se una parola facesse parte di un nome proprio, e ha interpretato male la struttura grammaticale. I problemi grammaticali sono più ovvii nell'esempio seguente. John ha trovato il maiale, o il maiale ha trovato John?

```
>>> babelize_shell()
Babel> The pig that John found looked happy
Babel> german
Babel> run
0> The pig that John found looked happy
1> Das Schwein, das John fand, schaute gl?cklich
2> The pig, which found John, looked happy
```

La traduzione meccanica è difficile perché la parola data potrebbe avere svariate possibili traduzioni (a seconda del suo significato), e perché l'ordine della parole deve essere cambiato per mantenere la struttura grammaticale della lingua d'arrivo. Oggi queste difficoltà si stanno affrontando raccogliendo massicce quantità di testi paralleli da giornali e siti web governativi che pubblicano documenti in due o più lingue. Avendo un documento in tedesco ed inglese, e possibilmente un dizionario bilingue, possiamo automaticamente accoppiare le frasi, un processo chiamato **allineamento di testo**. Una volta che abbiamo un milione o più coppie di frasi, possiamo analizzare parole ed espressioni corrispondenti, e costruire un modello che possa essere usato per tradurre un nuovo testo.

Sistemi di dialogo parlato

Nella storia dell'intelligenza artificiale, la misura principale dell'intelligenza è stata linguistica, ossia il **Test Turing**: può un sistema di dialogo, rispondendo al testo immesso dall'utente, agire così naturalmente da non distinguersi da una risposta generata da un umano? Al contrario, i sistemi di dialogo commerciale di oggi sono molto limitati, ma tuttavia eseguono funzioni utili in domini strettamente inquadrati, come possiamo vedere qui:

S: Come posso esserle d'aiuto?
U: Quando verrà proiettato Salvate il soldato Ryan?
S: In quale cinema?
U: Il cinema Paramount
S: Salvate il soldato Ryan non verrà proiettato al cinema Paramount, ma al cinema Madison alle 3:00, 5:30, 8:30 e 10:30.

Non potreste chiedere a questo sistema di fornire istruzioni di guida o dettagli su ristoranti a meno che le informazioni richieste siano già state immagazzinate e coppie di domande e risposte adeguate siano state incorporate nel sistema di elaborazione del linguaggio.

Osservate che questo sistema sembra capire gli obiettivi dell'utente: l'utente chiede quando un film verrà proiettato ed il sistema da ciò determina correttamente che l'utente vuole vedere il film. Questa deduzione sembra così ovvia che probabilmente non avrete notato che è stata creata, tuttavia un sistema di linguaggio naturale ha bisogno di essere dotato di questa capacità per interagire naturalmente. Senza di ciò, alla domanda *Sai quando trasmetteranno Salvate il soldato Ryan?,* un sistema potrebbe rispondere inutilmente con un freddo *Sì*. Tuttavia, gli sviluppatori di sistemi di dialogo commerciale usano supposizioni contestuali e logica d'affari per assicurare che i modi differenti in cui un utente possa esprimere richieste o fornire informazioni siano affrontate in un modo che abbia senso per la particolare applicazione. Quindi, se digitate *Quando è..., o Voglio sapere quando..., o Puoi dirmi quando...,* semplici regole porteranno sempre ad orari di programmazione. È sufficiente al sistema per fornire un servizio utile.

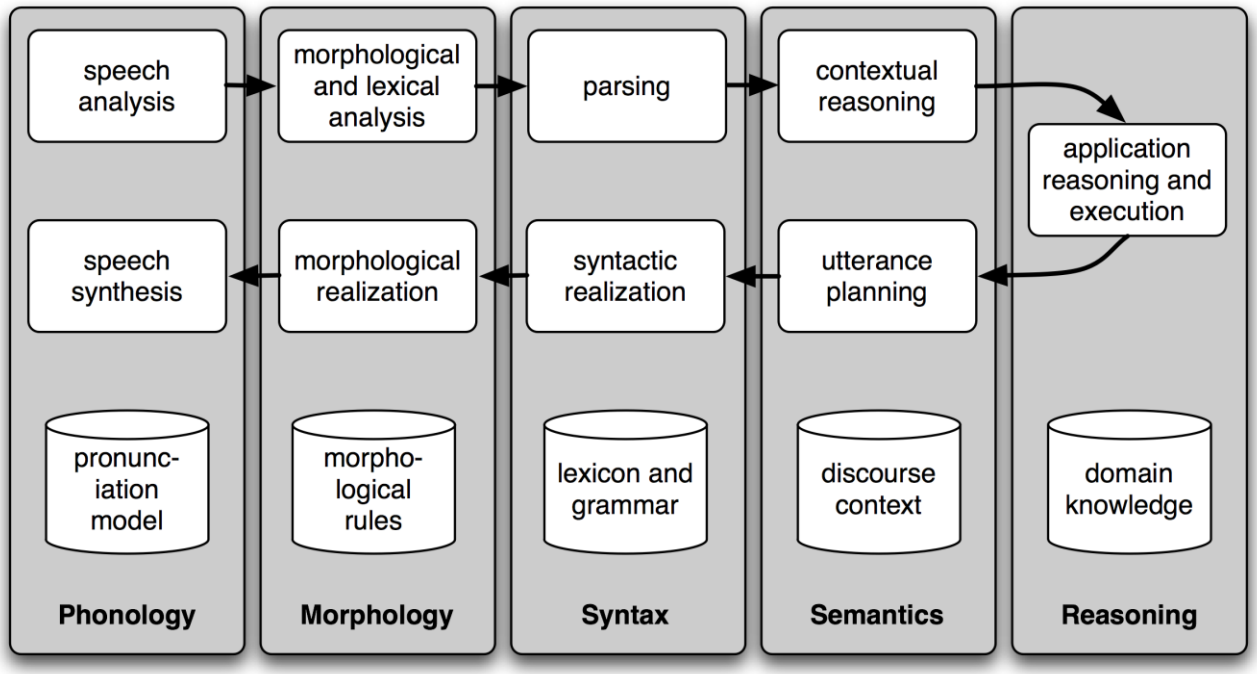


Figura 1.5: Semplice architettura a condotto per il Sistema di dialogo parlato: l'immissione del parlato (in alto a sinistra) è analizzata, le parole riconosciute, le frasi analizzate ed interpretate nel contesto, azioni specifiche dell'applicazione hanno luogo (in alto a destra); un responso è pianificato, realizzato come una struttura sintattica, poi alla parole opportunamente coniugate, e finalmente alla produzione del parlato; tipi differenti di comprensione linguistica prendono parte ad ogni fase del processo.

I sistemi di dialogo ci danno un'opportunità di accennare al condotto comunemente presunto per NLP. La Figura 1.5 mostra l'architettura di un semplice sistema di dialogo. In cima al diagramma, da sinistra a destra c'è un "condotto" di alcune **componenti** di comprensione del linguaggio. Formano una mappa dall'immissione del discorso attraverso la congiunzione sintattica fino a qualche tipo di rappresentazione del linguaggio. Nel mezzo, da sinistra a destra c'è il condotto contrario di componenti per convertire i concetti al discorso. Questi componenti creano gli aspetti dinamici del sistema. In fondo al diagramma ci sono alcuni corpi rappresentativi di informazioni statiche: i depositi di dati inerenti la lingua che i componenti d'elaborazione designano per svolgere il loro lavoro.

Nota

Il vostro turno: Per un esempio di un sistema di dialogo primitivo, provate ad avere una conversazione con un chatbot NLTK. Per vedere i chatbot disponibili, avviate `nltk.chat.chatbots()`. (Ricordate di fare prima `import nltk`).

Implicazione testuale

La sfida della comprensione del linguaggio è stata riportata in auge in anni recenti da una "mansione condivisa" pubblica chiamata Riconoscimento dell'Implicazione Testuale (RTE). Lo scenario di base è semplice. Supponete di voler trovare una prova che supporti l'ipotesi: *Sandra Goudie fu sconfitta da Max Purnell*, ed avete un altro breve testo che sembra essere rilevante, ad esempio, *Sandra Goudie fu eletta per la prima volta al parlamento alle elezioni del 2002, vincendo per un pelo la poltrona del Coromandel sconfiggendo il candidato laburista Max Purnell e spingendo l'incombente deputato dei Verdi Jeanette Fitzsimons in terza posizione*. Il testo fornisce abbastanza prove per farvi accettare l'ipotesi? In questo caso la risposta sarebbe "No." Potete trarre questa conclusione facilmente, ma è molto difficile trovare metodi automatici per prendere la decisione giusta. La sfida RTE forniscono dati che permettono ai competitori di sviluppare i loro sistemi, ma non abbastanza dati alle tecniche di apprendimento per macchine dalla "forza bruta" (un argomento che affronteremo nel sesto capitolo). Di conseguenza, un'analisi linguistica è cruciale. Nell'esempio precedente, è importante per il sistema notare che *Sandra Goudie* è il nome delle persona sconfitta nell'ipotesi, non la persona che attua la sconfitta nel testo. Come altra illustrazione della difficoltà del compito, considerate la seguente coppia di testo ed ipotesi:

(7) a. Testo: David Golinkin è l'editore o autore di diciotto libri, ed oltre 150 responsi, articoli, sermoni e libri
b. Ipotesi: Golinkin ha scritto diciotto libri

Per determinare se l'ipotesi è supportata dal testo, il sistema necessita la conoscenza della seguente situazione: (i) se qualcuno è un autore di un libro, allora lui/lei hanno scritto quel libro; (ii) se qualcuno è un editore di un libro, allora lui/lei non ha scritto (tutto) il libro; (iii) se qualcuno è editore o autore di diciotto libri, allora non si può concludere che lui/lei è autore di diciotto libri.

Limitazioni di NLP

Nonostante la ricerca avanzi in compiti come RTE, i sistemi del linguaggio naturale che sono stati impiegati per applicazioni reali non possono ancora svolgere ragionamenti di senso comune o aspirare alla comprensione del mondo in maniera generale e robusta. Possiamo aspettare che questi difficili problemi di intelligenza artificiale vengano risolti, ma non frattempo è necessario vivere con alcune gravi limitazioni sulla capacità di ragionamento e comprensione dei sistemi di linguaggio naturale. Di conseguenza, sin dall'inizio un importante scopo della ricerca NLP è stato fare progressi sul difficile compito di costruire tecnologie che "comprendano il linguaggio", usando tecniche superficiali ma potenti invece di capacità di comprensione e ragionamento illimitate. È proprio questo uno degli obiettivi di questo libro, e speriamo di attrezzarvi con la conoscenza e le abilità necessarie per costruire sistemi NLP utili, e contribuire all'aspirazione di costruire macchine intelligenti.

2 Accesso ai *corpora text* e alle risorse lessicali.

Praticamente lavorare con il Natural Language Processing usualmente usa i classici bodies con dati linguistici o corpora. Questo capitolo vuole rispondere alle seguenti domande:

1. quali sono alcuni corpora testuali e le risorse lessicali utili e come possiamo averne accesso tramite Python?
2. Quali costrutti di Python sono più utili per questo lavoro?
3. Come evitiamo di ripeterci quando scriviamo in codice Python?

Questo capitolo continua a presentare i concetti di programmazione attraverso esempi, nel contesto di un obiettivo di elaborazione linguistica. Vedremo più avanti ogni costrutto di Python in modo sistematico. Non vi preoccupate se vedete un esempio che contiene elementi che non vi sembrano familiari, provate e vedete cosa succede, e modificarlo sostituendo alcune parti del codice con un testo diverso o parole diverse. Questo vorrebbe dire che associate un lavoro ad una espressione idiomatica di programmazione, e imparare i come e i perché successivamente.

2.1 Accesso ai *corpora text*

come abbiamo precedentemente detto, un corpus testuale è un ambio body testuale. Molti *corpora* sono pensati per contenere un bilancio sottile di elementi in uno o più generei. Abbiamo preso in esame piccole collezioni testuali nel Capitolo 1, come i discorsi chiamati “Us Presidential Inaugural Addresses”. Questo corpo testuale in particolare in effetti contiene dozzine di testi individuale, una per ogni discorso, ma per convenienza le abbiamo unite e considerate un unico testo. Il Capitolo 1 usava anche diversi testi predefiniti a cui avevamo accesso scrivendo from book import *. Comunque dato che vogliamo essere in grado di lavorare con altri testi, questa sezione prende in esame un varietà di testi. Vedremo come selezionare un testo individuale e come lavorare con ciascuno di essi.

Gutenberg Corpus

NLTK include una piccola sezione di testi dal Project Gutenberg, un archivio testuale elettronico, che contiene qualcosa come 25 mila libri gratuiti in formato digitali, condivisi attraverso la pagina http://www.gutenberg.org/. Vogliamo iniziare importando su Python il pacchetto di NLTK, poi chiedergli di guardare nltk.corpus.guntenberg.fileids (), il file si indenfica nel modo seguente:

```
>>> 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']
```

Riprendiamo uno dei primi testi – *Emma* di Jane Austen – e diamogli un nome breve, emma, cerchiamo di trovare quante parole contenga

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

Nota

Nella Sezione 1.1, abbiamo mostrato come potevamo portare la concordanza di un tesco come text1 con il comando text.1.concordance (). Comunque, questo presuppone che stiamo usando uno dei nove testi ottenuti come risultato del comando from nltk.book import *. Ora ciò che avete iniziato esaminando i dati da nltk.corpus, come nell'esempio precedente, dovete usare i seguenti due istruzioni per fare una concordanza e altri lavori dalla Sezione 1.1:

```
>>> emma = nltk.Text(nltk.corpus.gutenberg.words('austen-emma.txt'))
>>> emma.concordance("surprize")
```

Quando definiamo emma invochiamo le funzioni words() dell'oggetto gutenberg nel pacchetto dei corpus di NLTK. Ma dato che è poco maneggevole scrivere dei nomi così lunghi ogni volta, Python ci dà un'altra versione del comando import come vediamo:

```
>>> from nltk.corpus import gutenberg
>>> gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', ...]
>>> emma = gutenberg.words('austen-emma.txt')
```

scriviamo un piccolo programma che mostri altra informazione per ciascuno dei testi facendo un loop di tutti i valori di fileid corrispondente ai file identificatori di gutenberg elencati prima e facendo delle statistiche per ciascun testo. Per una visualizzazione compatta, ci assicureremo che i numeri siano interi usando il comando int ().

```
>>> 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 int(num_chars/num_words), int(num_words/num_sents), int(num_words/num_vocab), fileid
...
4 21 26 austen-emma.txt
4 23 16 austen-persuasion.txt
4 24 22 austen-sense.txt
4 33 79 bible-kjv.txt
4 18 5 blake-poems.txt
4 17 14 bryant-stories.txt
4 17 12 burgess-busterbrown.txt
4 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.txt
4 18 24 edgeworth-parents.txt
4 24 15 melville-moby_dick.txt
4 52 10 milton-paradise.txt
4 12 8 shakespeare-caesar.txt
4 13 7 shakespeare-hamlet.txt
4 13 6 shakespeare-macbeth.txt
4 35 12 whitman-leaves.txt
```

Questo programma mostra tre valori statistici per ogni testo: media della lunghezza delle parole, media della lunghezza delle frasi, numero di volte che ogni parola compariva nel testo in media (il nostro punteggio di diversità lessicale). Osservate la media della lunghezza delle parole, sembrerebbe una proprietà dell'inglese, dal momento che il valore è 4. (di fatti la media della lunghezza delle parole sarebbe 3 e non 4, dal momento che la variabile num_chars conta gli spazi) Da contrasto la media della lunghezza delle frasi e della diversità lessicale sembrerebbe una caratteristica di ciascun autore.

L'esempio precedente mostra anche come possiamo accedere al testo "raw", non suddiviso in parti. La funzione raw () ci dà il contenuto del file senza elaborazione linguistica. Così per esempio, len(gutenberg.raw('blake-poems.txt') ci dice quante *lettere* appaiono nel testo, includendo gli spazi tra le parole. La funzione sents() divide il testo in più 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[1037]
['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', ...], ...]
```

Nota

La maggior parte dei lettori di corpi NLTK include una varietà di metodi di accesso oltre words (), raw () oppure sents (). Un contenuto linguistico più vasto è visibile in qualche corpora, come in quelli

chiamati parti-di-un-discorso, i dialoghi, gli alberi sintattici e così via; vedremo queste cose nei capitoli successivi.

Web e Chat Text

Anche se il progetto Guntenberg contiene migliaia di libri, rappresenta una letteratura prestabilita. È importante considerare anche un linguaggio meno formale. Una piccola collezione di testi web di NLTK include dei contenuti da discussioni sui forum di Firefox, conversazioni di New York, la sceneggiatura di *La maledizione della prima luna* (*Pirates of the Carribean*), 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...
```

ci sono anche un corpus di sessioni di chat, collezionate originalmente dal Naval Postgraduate School per ricerche di rilevamento automatico dei predatori Internet.

Il corpus comprende più di 10 mila post, resi anonimi dal rimpiazzo degli username con generici nomi dal "UserNNN", e manualmente modificati per rimuovere ogni tipo di informazione sull'identità degli utenti. Il corpus è organizzato in 15 file, dove ciascuno contiene diverse centinaia raccolte in una data prestabilita da chatroom di età specifica (adolescenti, ventenni, trentenni, quarantenni, più diverse chatroom generiche per adulti). Il nome del file contiene la data, la chatroom, il numero di post; ad esempio 10-19-20s_706posts.xml contiene 706 post generati dalla chat ventenni il 10/19/2006.

```
>>> 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', '.']
```

Brown Corpus

Il Brown Corpus è stato il primo corpo elettronico da un milione di parole inglese, creato nel 1961 alla Brown University. Questo corpo contiene testi da 500 fonti, le fonti sono ordinate secondo il genere, come *news, editoriali*, e così via. La tavola 2.1 ci mostra un esempio di ciascun genere (per la lista completa andare su http://icame.uib.no/brown/bcm-los.html).

Tavola 2.1:

esempio di documento dal ogni sezione del Brown Corpus

ID	File	Genre	Description
A16	ca16	news	Chicago Tribune: *Society Reportage*
B02	cb02	editorial	Christian Science Monitor: *Editorials*
C17	cc17	reviews	Time Magazine: *Reviews*
D12	cd12	religion	Underwood: *Probing the Ethics of Realtors*
E36	ce36	hobbies	Norling: *Renting a Car in Europe*
F25	cf25	lore	Boroff: *Jewish Teenage Culture*
G22	cg22	belles_lettres	Reiner: *Coping with Runaway Technology*
H15	ch15	government	US Office of Civil and Defence Mobilization: *The Family Fallout Shelter*
J17	cj19	learned	Mosteller: *Probability with Statistical Applications*
K04	ck04	fiction	W.E.B. Du Bois: *Worlds of Color*
L13	cl13	mystery	Hitchens: *Footsteps in the Night*
M01	cm01	science_fiction	Heinlein: *Stranger in a Strange Land*
N14	cn15	adventure	Field: *Rattlesnake Ridge*
P12	cp12	romance	Callaghan: *A Passion in Rome*
R06	cr06	humor	Thurber: *The Future, If Any, of Comedy*

Possiamo accedere al corpus come lista di parole o lista di frasi (dove ogni frase è essa stessa una lista di parole). Possiamo anche scegliere categorie particolare o file da leggere:

```
>>> from nltk.corpus import brown
>>> brown.categories()
['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies',
'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance',
'science_fiction']
>>> brown.words(categories='news')
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
>>> brown.words(fileids=['cg22'])
['Does', 'our', 'society', 'have', 'a', 'runaway', ',', ...]
>>> brown.sents(categories=['news', 'editorial', 'reviews'])
[['The', 'Fulton', 'County'...], ['The', 'jury', 'further'...], ...]
```

Il Brown Corpus è una risorsa conveniente per studiare le differenze sistematiche attraverso i generi, una sorta di indagine linguistica chiamata *stilistica*. Compariamo diversi generi suoi loro *modal verbs*. Il primo passo è introdurre i conteggi per un genere particolare. Ricordate di fare import nltk prima di fare ciò che 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
```

Nota

Sta a voi: Scegliete una diversa sezione del Brown Corpus e adattate l'esempio precedente per contare una selezione di parole con *wh*, come *what*, *where*, *who* e *why*.

Successivamente dobbiamo ottenere i conteggi per ogni genere a cui siamo interessati. Usiamo il supporto di NLKT per la frequenza di distribuzione condizionale. Queste sono presentate in modo sistematico nella Sezione 2.2 dove è anche "disfatto" il seguente codice linea per linea. Peri l momento potete ignorare i dettagli e concentrarvi 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   43
          humor   16   30    8    8    9   13
```

Osservate che il *modal verb* più usato nel genere delle news è *will*, mentre quello più usato nei "romantico" è *could*. Lo avreste mai detto? L'idea che il contatore di parole possa distinguere i genere sarà ripresa in chap-data-intensive.

Reuters Corpus

Il Reuters Corpus contiene 10.788 documenti di notizie che totalizzano 1.3 milioni di parole. Il documento è stato ordinato in 90 argomenti, raggruppato in due insieme chiamati "training" e "test"; così il testo con il fileid 'test/14826' è un documento preso dall'insieme "test". Questa suddivisione è dato dagli algoritmi di "training" e "testing" e determinano automaticamente l'argomento di un documento, come vedremo nel chap-data-intensive.

```
>>> 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', ...]
```

A differenza del Brown Corpus, le categorie nel Reuters Corpus si sovrappongono l'un l'altra, semplicemente perché nuove storie spesso coprono diversi insiemi. Possiamo chiedere anche per gli

insiemi usati da uno o più documenti oppure se il documento include una o più categorie. Per semplicità, il metodo del corpus accetta o un singolo fileid o una lista di fileids.

```
>>> reuters.categories('training/9865')
['barley', 'corn', 'grain', 'wheat']
>>> reuters.categories(['training/9865', 'training/9880'])
['barley', 'corn', 'grain', 'money-fx', 'wheat']
>>> reuters.fileids('barley')
['test/15618', 'test/15649', 'test/15676', 'test/15728', 'test/15871', ...]
>>> reuters.fileids(['barley', 'corn'])
['test/14832', 'test/14858', 'test/15033', 'test/15043', 'test/15106',
'test/15287', 'test/15341', 'test/15618', 'test/15618', 'test/15648', ...]
```

In modo simile possiamo richiedere specifiche parole o frasi che vogliamo in termini di file o categorie. La prima manciata di parole di ognuno di questi testi sono i titoli, che per convenzione vengono memorizzati in maiuscolo.

```
>>> reuters.words('training/9865')[:14]
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', 'BIDS',
'DETAILED', 'French', 'operators', 'have', 'requested', 'licences', 'to', 'export']
>>> reuters.words(['training/9865', 'training/9880'])
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
>>> reuters.words(categories='barley')
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
>>> reuters.words(categories=['barley', 'corn'])
['THAI', 'TRADE', 'DEFICIT', 'WIDENS', 'IN', 'FIRST', ...]
```

Inaugural Address Corpus

Nella Sezione 1.1, abbiamo visto l'Inaugural Address Corpus, ma l'abbiamo considerato come un testo singolo. Il grafico in fig-inaugural usa i "word offset" come uno degli assi; questo è l'indice numerico delle parole nel corpus, contando dalla prima parola del primo discorso. Tuttavia, il corpus è una raccolta di 55 testi, uno per ogni discorso 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', ...]
```

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

Guardiamo a come le parole *America* e *citizen* sono usate più volte. Il codice seguente converte le parole dell'Inaugural Corpus in minuscolo usando w.lower (), poi controlliamo se abbiamo iniziato con quale dei due "target", *america* o *citizen* usando startswith (). Così conteremo parole come *American's* e *Citizens*. Vedremo la distribuzione con frequenza condizionale nella Sezione 2.2; per il momento considerate l'output, mostrato nella Figura 2.1.

```
>>> 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()
```

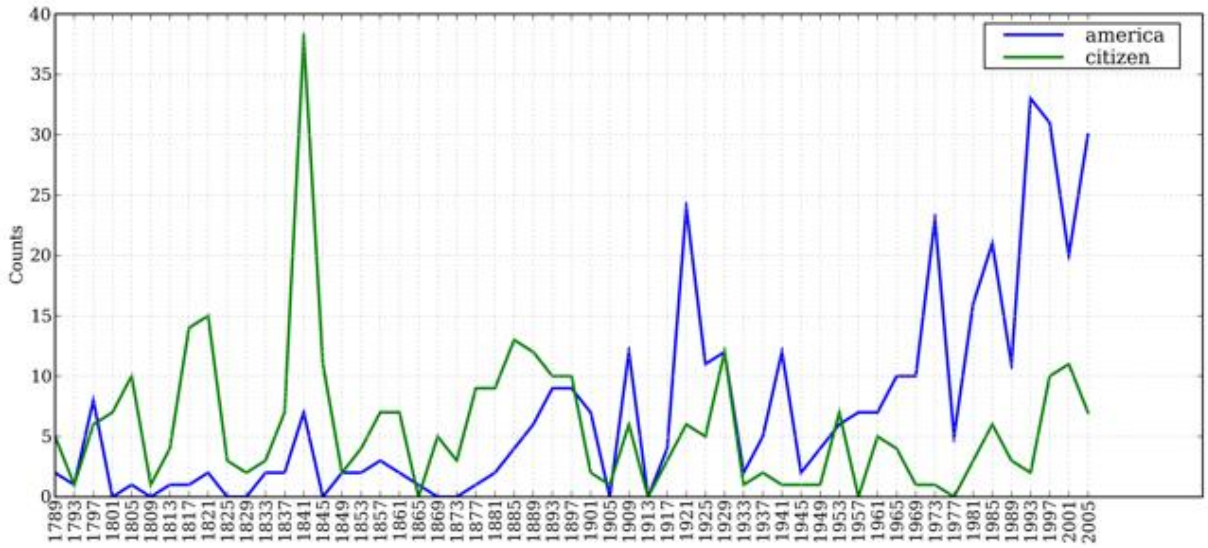


Figura 2.1: Plot di una distribuzione a frequenza condizionale: tutte le parole dell'Inaugural Address Corpus che iniziano con *america* o *citizen* sono nel conteggio; i conteggi restano separati per ogni discorso; sono tracciati in modo tale che possa essere osservato il trend nel passare del tempo; i conteggi non sono normalizzati per la lunghezza del documento.

Annotated Text Corpora

Molti corpora testuali contengono annotazioni linguistiche, rappresentate nelle etichette POS, entità, strutture sintattiche, ruoli semantici, e così via. NLTK provvede a dare modi convenienti per accedere a diversi di questi corpora, e ha pacchetti di dati contenenti esempi di corpora e di corpi, scaricabili gratuitamente per uso nell'insegnamento e nella ricerca. Nella Tavola 2.2 possiamo vedere la lista di alcuni corpora. Per informazione sul loro download andate su http://www.nltk.org/data. Per maggiori esempi su come accedere ai corpora NLTK, consultate il Corpus HOWTO sul http://www.nltk.org/howto.

Tavola 2.2:

Alcuni esempi di Corpora e Corpus distribuiti con NLTK: per informazione su come farne il download, prego consultare il sito di NLTK.

Corpus	Compiler	Contents
Brown Corpus	Francis, Kucera	15 genres, 1.15M words, tagged, categorized
CESS Treebanks	CLiC-UB	1M words, tagged and parsed (Catalan, Spanish)
Chat-80 Data Files	Pereira & Warren	World Geographic Database
CMU Pronouncing Dictionary	CMU	127k entries
CoNLL 2000 Chunking Data	CoNLL	270k words, tagged and chunked
CoNLL 2002 Named Entity	CoNLL	700k words, pos- and named-entity-tagged (Dutch, Spanish)
CoNLL 2007 Dependency Treebanks (sel)	CoNLL	150k words, dependency parsed (Basque, Catalan)
Dependency Treebank	Narad	Dependency parsed version of Penn Treebank sample
Floresta Treebank	Diana Santos et al	9k sentences, tagged and parsed (Portuguese)
Gazetteer Lists	Various	Lists of cities and countries
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Gutenberg (selections)	Hart, Newby, et al	18 texts, 2M words
Inaugural Address Corpus	CSpan	US Presidential Inaugural Addresses (1789-present)
Indian POS-Tagged Corpus	Kumaran et al	60k words, tagged (Bangla, Hindi, Marathi, Telugu)
MacMorpho Corpus	NILC, USP, Brazil	1M words, tagged (Brazilian Portuguese)
Movie Reviews	Pang, Lee	2k movie reviews with sentiment polarity classification
Names Corpus	Kantrowitz, Ross	8k male and female names
NIST 1999 Info Extr (selections)	Garofolo	63k words, newswire and named-entity SGML markup
NPS Chat Corpus	Forsyth, Martell	10k IM chat posts, POS-tagged and dialogue-act tagged
PP Attachment Corpus	Ratnaparkhi	28k prepositional phrases, tagged as noun or verb modifiers
Proposition Bank	Palmer	113k propositions, 3300 verb frames
Question Classification	Li, Roth	6k questions, categorized
Reuters Corpus	Reuters	1.3M words, 10k news documents, categorized
Roget's Thesaurus	Project Gutenberg	200k words, formatted text
RTE Textual Entailment	Dagan et al	8k sentence pairs, categorized
SEMCOR	Rus, Mihalcea	880k words, part-of-speech and sense tagged
Senseval 2 Corpus	Pedersen	600k words, part-of-speech and sense tagged
Shakespeare texts (selections)	Bosak	8 books in XML format
State of the Union Corpus	CSPAN	485k words, formatted text
Stopwords Corpus	Porter et al	2,400 stopwords for 11 languages
Swadesh Corpus	Wiktionary	comparative wordlists in 24 languages
Switchboard Corpus (selections)	LDC	36 phonecalls, transcribed, parsed
Univ Decl of Human Rights	United Nations	480k words, 300+ languages
Penn Treebank (selections)	LDC	40k words, tagged and parsed
TIMIT Corpus (selections)	NIST/LDC	audio files and transcripts for 16 speakers
VerbNet 2.1	Palmer et al	5k verbs, hierarchically organized, linked to WordNet
Wordlist Corpus	OpenOffice.org et al	960k words and 20k affixes for 8 languages
WordNet 3.0 (English)	Miller, Fellbaum	145k synonym sets

Corpora in altre lingue

NLTK porta corpora di molte lingue diverse, in alcuni casi sarà necessario imparare a manipolare i character encodings in Python prima di usare questi corpora (vedere la Sezione 3.3).

```
>>> 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')
['\xe0\xa4\xaa\xe0\xa5\x82\xe0\xa4\xb0\xe0\xa5\x8d\xe0\xa4\xa3',
'\xe0\xa4\xaa\xe0\xa5\x8d\xe0\xa4\xb0\xe0\xa4\xa4\xe0\xa4\xbf\xe0\xa4\xac\xe0\xa4
\x82\xe0\xa4\xa7', ...]
>>> 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:]
[u'Saben', u'umat', u'manungsa', u'lair', u'kanthi', ...]
```

L'ultimo di questi corpora, udhr, contiene la Dichiarazione Universale dei Diritti Umani in più di 300 lingue. I fileids di questo corpus includono informazione sui character encoding usati nel file, come UTF8 o LATIN1. Usiamo un condizionale di frequenza di distribuzione per esaminare le differenze delle lunghezze delle parole in una selezione di lingue includendo il corpus udhr. L'output è mostrato nella Figura 2.2 (fate partire il programma voi stessi per vedere l'intreccio di colori). Fate attenzione al fatto che false e true sono valori built-in di Python.

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

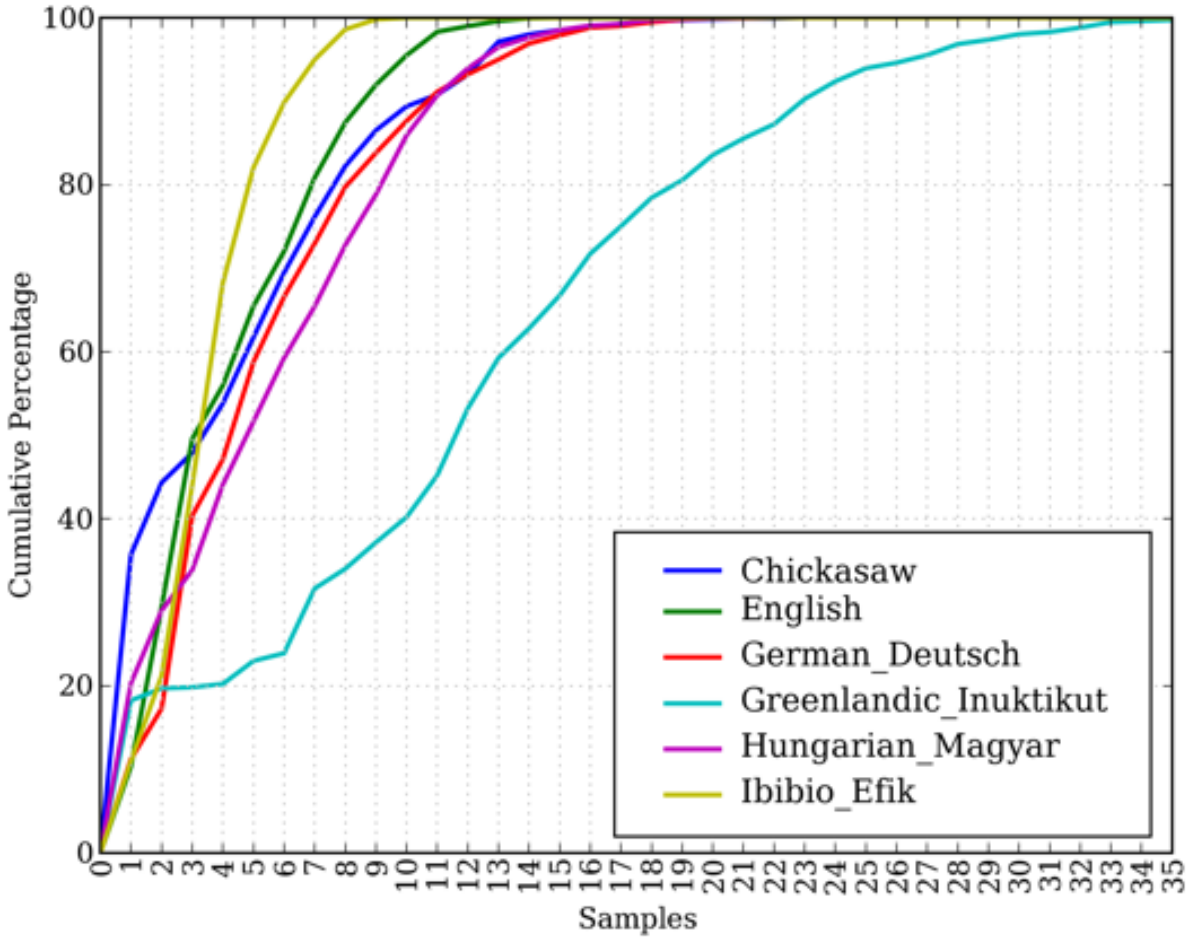


Figura 2.2: Distribuzione di Lunghezza di Parole Cumulativa: Sei traduzioni della Dichiarazione Universale dei Diritti Umani sono state incluse nel processo; questo grafico mostra che parole che hanno 5 o meno lettere sono circa l'80% per il testo Ibibio, il 60% se in tedesco, il 25% se in Inuktitut.

Nota

Sta a voi: prendete una lingua di interesse nel udhr.fileids() , e definite la variabile raw_text = udhr.raw *(Language-Latin1).* Ora intrecciate la frequenza di distribuzione delle lettere o dei testi usando nltk.FreqDist(raw_text).plot().

Sfortunatamente, molte lingue, molti corpora non sono ancora disponibili. Spesso c'è uno scarso supporto governativo o industriale nello sviluppo di risorse linguistiche e gli sforzi individuali sono frammentari, difficili da scoprire o da riutilizzare. Alcune lingue non hanno sistemi di scrittura stabiliti o sono in via di estinzione. (guarda la Sezione 2.7 per suggerimenti su come trovare risorse linguistiche).

Stuttura testuale dei Corpus

Abbiamo visto una varietà di strutture dei corpus, queste sono riassunte nella Figura 2.3. Il tipo più semplice è privo di ogni struttura. È giusto una collezione di testi. Spesso i testi sono raggruppati in categorie che possono corrispondere a genere, origine, autore, lingua etc. Alcune volte queste categorie si sovrappongono, come si può vedere nel caso delle categorie tematiche, così un testo può essere rilevante per più di un argomento. Raramente i le collezioni testuali hanno una struttura temporale le raccolte di notizie sono l'esempio più comuni.

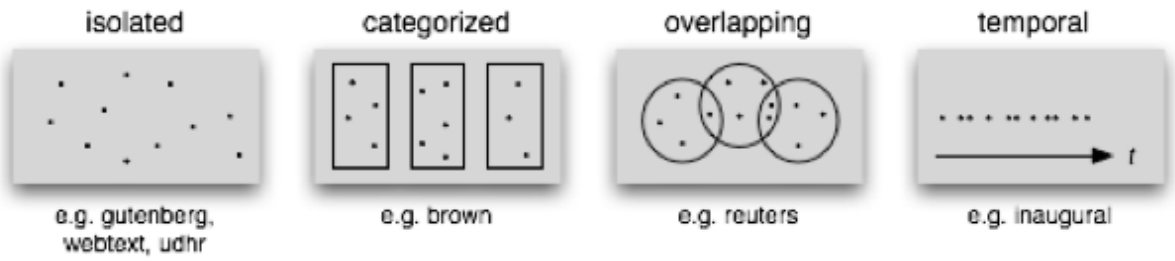


Figura 2.3: Strutture ricorrenti per i Text Corpora: i tipi più semplici di corpora sono una collezione di testi isolati con nessuna organizzazione particolare; alcuni corpora sono strutturati secondo categorie come il genere (il Brown Corpus), alcune categorizzazioni si sovrappongono, come le categorie testuali (Reuters Corpus), altri corpora rappresentano l'uso del linguaggio attraverso il tempo (Inaugural Address Corpus).

Tavola 2.3:

Funzionalità base delcorpus definita in NLTK: maggiore documentazione può essere trovata usando help (nltk.corpus.reader) e leggendo online il corpus HOWTO al http://www.nltk.org/howto

Example	Description
`fileids()`	the files of the corpus
`fileids([categories])`	the files of the corpus corresponding to these categories
`categories()`	the categories of the corpus
`categories([fileids])`	the categories of the corpus corresponding to these files
`raw()`	the raw content of the corpus
`raw(fileids=[f1,f2,f3])`	the raw content of the specified files
`raw(categories=[c1,c2])`	the raw content of the specified categories
`words()`	the words of the whole corpus
`words(fileids=[f1,f2,f3])`	the words of the specified fileids
`words(categories=[c1,c2])`	the words of the specified categories
`sents()`	the sentences of the whole corpus
`sents(fileids=[f1,f2,f3])`	the sentences of the specified fileids
`sents(categories=[c1,c2])`	the sentences of the specified categories
`abspath(fileid)`	the location of the given file on disk
`encoding(fileid)`	the encoding of the file (if known)
`open(fileid)`	open a stream for reading the given corpus file
`root()`	the path to the root of locally installed corpus
`readme()`	the contents of the README file of the corpus

Il supporto dei reader dei corpus di NLTK accedono efficientemente a una varietà di corpora, e possono essere usati per lavorare con nuovi corpora. La Tavola 2.3 mostra una lista procurata attraverso i lettori di corpus. Illustriamo le differenze tra alcuni degli metodi di accesso al corpus qui di seguito:

```
>>> raw = gutenberg.raw("burgess-busterbrown.txt")
>>> raw[1:20]
'The Adventures of B'
>>> words = gutenberg.words("burgess-busterbrown.txt")
>>> words[1:20]
['The', 'Adventures', 'of', 'Buster', 'Bear', 'by', 'Thornton', 'W', '.',
'Burgess', '1920', ']', 'I', 'BUSTER', 'BEAR', 'GOES', 'FISHING', 'Buster',
'Bear']
>>> sents = gutenberg.sents("burgess-busterbrown.txt")
>>> sents[1:20]
[['I'], ['BUSTER', 'BEAR', 'GOES', 'FISHING'], ['Buster', 'Bear', 'yawned', 'as',
'he', 'lay', 'on', 'his', 'comfortable', 'bed', 'of', 'leaves', 'and', 'watched',
'the', 'first', 'early', 'morning', 'sunbeams', 'creeping', 'through', ...], ...]
```

Aprire il vostro Corpus

Se avete una vostra collezione di file di testo a cui volete avere accesso tramite uno dei metodi sopra elencati, potete semplicemente caricarli con l'aiuto del PalintextCorpusReader di NLTK. Controllate dove avete salvato i vostri file sul vostro file system; nell'esempio seguente vi abbiamo mandato sulla seguente directory /usr/share/dict. Qualsiasi sia la location, usate questo come valore corpus_root. Il secondo paramento dell'initializer PalintextCorpusReader può essere una lista di fileids, come ['a.txt', 'test/b.txt'] , oppure un pattern che faccia un match di tutti i fileids, come '[abc]/.*\.txt' (guardate la Sezione 3.4 per informazione sulle espressioni regolari).

```
>>> from nltk.corpus import PlaintextCorpusReader
>>> corpus_root = '/usr/share/dict'
>>> wordlists = PlaintextCorpusReader(corpus_root, '.*')
>>> wordlists.fileids()
['README', 'connectives', 'propernames', 'web2', 'web2a', 'words']
>>> wordlists.words('connectives')
['the', 'of', 'and', 'to', 'a', 'in', 'that', 'is', ...]
```

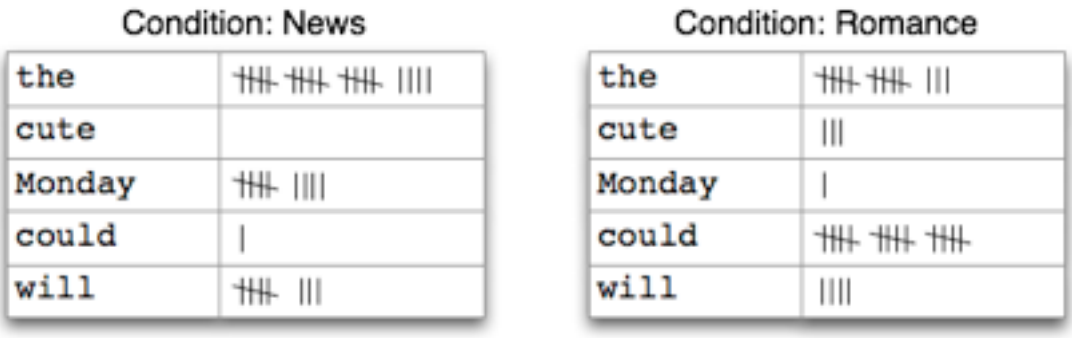
Un altro esempio, supponiamo che avete una vostra copia locale di Penn Treebank (release 3), in c:\corpora. Possiamo usare il BracketParseCorpusReader per accedere al corpus. Possiamo specificare la location con corpus_root della sezione del Wall Street Journal del corpus, e dare un file_pattern che faccia un match dei file contenuti in sottocartelle (usando gli slash).

```
>>> from nltk.corpus import BracketParseCorpusReader
>>> corpus_root = r"C:\corpora\penntreebank\parsed\mrg\wsj"
>>> file_pattern = r".*/wsj_.*\.mrg"
>>> ptb = BracketParseCorpusReader(corpus_root, file_pattern)
>>> ptb.fileids()
['00/wsj_0001.mrg', '00/wsj_0002.mrg', '00/wsj_0003.mrg', '00/wsj_0004.mrg', ...]
>>> len(ptb.sents())
49208
>>> ptb.sents(fileids='20/wsj_2013.mrg')[19]
['The', '55-year-old', 'Mr.', 'Noriega', 'is', "n't", 'as', 'smooth', 'as', 'the',
'shah', 'of', 'Iran', ',', 'as', 'well-born', 'as', 'Nicaragua', "'s", 'Anastasio',
'Somoza', ',', 'as', 'imperial', 'as', 'Ferdinand', 'Marcos', 'of', 'the', 'Philippines',
'or', 'as', 'bloody', 'as', 'Haiti', "'s", 'Baby', Doc', 'Duvalier', '.']
```

2.2 Conditional Frequency Distributions

abbiamo introdotto la frequency distribuitons nella Sezione 1.3. abbiamo visto come dando alcune liste mylist di parole o altri elementi, FreqDist (mylist) avrebbe calcolato il numero di occorrenze per ogni elemento nella lista. Qui generalizzeremo questa idea.

Quando il testo di un corpus è diviso in diverse categorie, di genere, tema, autore etc, possiamo mantenere separata la frequency distributions per ogni categoria. Questo ci permetterà di studiare le differenze sistematiche tra le categorie. Nella sezione precedente abbiamo raggiunto ciò usando il ConditionalFreqDist di NLTK. Un *conditional frequency distribution* è una collezione di distribuzioni di freqeunza, ognuna della quali per una diversa "condizione". La condizione sarà spesso la categoria del testo. La Figura 2.4 raffigura un frammento del *conditional frequency distribution* dando due condizioni, una per le notizie e un par i testi romanzeschi.



Condition: News

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

Condition: Romance

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

Figura 2.4: Conteggio delle parole che appaiono in una collezione di testi (*conditional frequency distribution).*

Condizioni e Eventi

I conteggi di *frequenci distribution* osserva gli eventi, come la presenza delle parole in un testo. Un *conditional frequency distribution* ha bisogno di un'associazione di ciascun evento con una condizione. Così invece di fare un processo di una sequenza di parole, abbiamo un processo di sequenze di coppie.

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

Ogni coppia ha la forma (condition, event) . Se facciamo un processo di tutto il Brown Corpus secondo i generi, avremmo 15 condizioni (una per ogni genere) e 1.161.192 eventi (uno per ciascuna parola).

Conteggio delle parole per genere

Nella Sezione 2.1 abbiamo visto il *conditional frequency distribution* in cui la condizione era la sezione del Brown Corpus e per ciascuna condizione abbiamo fatto il conteggio delle parole. Mentre la FreqDist () prende una lista semplice come input, il ConditionalFreqDist() prende una lista di coppie.

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

Facciamo una scomposizione, e guardiamo solo a due generi, notizie e romanzo. Per ogni genere, abbiamo fatto un ciclo su ogni parola nel genere, producendo coppie genere-parola.

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

Così, come possiamo vedere sotto, coppie all'inizio della lista genre_word saranno nella forma ('news', *word*), mentre quelle alla fine saranno ('romance', *word*).

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

Possiamo usare la lista di coppie per creare un ConditionalFreqDist, e salvarlo in una variabile cfd. Come al solito, possiamo scrivere il nome della variabile per ispezionarlo, e verificare se ha le due condizioni:

```
>>> cfd = nltk.ConditionalFreqDist(genre_word)
>>> cfd
<ConditionalFreqDist with 2 conditions>
>>> cfd.conditions()
['news', 'romance'] # [_conditions-cfd]
```

Accediamo alle due condizioni, e accertiamoci che ciascuna è solo una distribuzione di frequenza:

```
>>> cfd['news']
<FreqDist with 100554 outcomes>
>>> cfd['romance']
<FreqDist with 70022 outcomes>
>>> list(cfd['romance'])
[',', '.', 'the', 'and', 'to', 'a', 'of', '``', "''", 'was', 'I', 'in', 'he', 'had',
'?', 'her', 'that', 'it', 'his', 'she', 'with', 'you', 'for', 'at', 'He', 'on', 'him',
'said', '!', '--', 'be', 'as', ';', 'have', 'but', 'not', 'would', 'She', 'The', ...]
>>> cfd['romance']['could']
193
```

Plotting and Tabulating Distributions

Oltre a combinare due o più distribuzioni di frequenza, e di essere facile da inizializzare, un ConditionalFreqDist fornisce alcuni metodi utili per la tabulazione e la stampa.

L'intreccio della Figura 2.1 si basa su un *conditional frequency distribution*, riprodotto qui sotto. La condizione è degli elementi *america* o *citizen*, e i conteggi in output sono i numeri delle volte che la parola appare in un discorso particolare. Mostra il fatto che il nome del file per ciascun discorso, ad esempio 1865-Lincoln.txt contiene l'anno così come il primo dei quattro personaggi. Questo codice genera coppie ('america', '1865') per ogni singola istanza di una parola che è scritta in minuscolo e inizia con *america* – come *Americans* – nel file 1865-Lincoln.txt.

```
>>> from nltk.corpus import inaugural
>>> 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))
```

La Figura 2.2 è anch'essa basata su un *conditional frequency distribution*, riprodotto di seguito. Questa volta, la condizione è il nome della lingua e il conteggio deriva dalla lunghezza delle parole. Mostra che il nome del file per ogni lingua è il nome della lingua seguito da '-Latin1'(il character encoding).

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

Nei metodi plot () e tabulate () , possiamo specificare facoltativamente quali condizioni mostrare con un parametro tipo conditions=. Quando lo omettiamo, abbiamo tutte le condizioni. In modo simile, possiamo limitare gli esempi da mostrare utilizzando il parametro samples=. Questo ci permette di caricare una grande quantità di dati nel *conditional frequency distribution* e esplorarlo attravero il plotting o il tabulating con condizioni e esempi selezionati. Ci dà anche pieno controllo delle condizioni e esempi in ogni visualizzazione. Per esempio, possiamo fare una tabella della frequenza cumulativa dei dati solo per due lingue, e per parole non più lunghe di 10 caratteri, come mostrato di sotto. Possiamo interpretare l'ultima cella nella prima riga che mostra 1,638 parole del testo in inglese hanno 9 o meno lettere.

```
>>> cfd.tabulate(conditions=['English', 'German_Deutsch'],
...              samples=range(10), cumulative=True)
                  0    1    2    3    4    5    6    7    8    9
       English    0  185  525  883  997 1166 1283 1440 1558 1638
German_Deutsch    0  171  263  614  717  894 1013 1110 1213 1275
```

Nota

Sta a voi: lavorare con le notizie o i romanzi come generi dal Brown Corpus, trovare quali giorni della settimana sono più interessanti e quali sono più "romantici". Definire la variabile chiamata days contenente una lista dei giorni, ad esempio ['Monday', ...] . Ora tabulare il conteggio per queste parole usando cfd.tabulate(samples=days) . Ora cercare di fare la stessa cosa usando al posto di plot . Puoi controllare l'ordine dell'output dei giorni con l'aiuto di un parametro extra: conditions=['Monday', ...]

Potete notare come le espressioni su più linee sono state usate con i *conditional frequency distribution* e come sembrino liste di comprensione, ma senza le parentesi. In generale, quando usiamo una lista di comprensione come un parametro per una funzione, come set([w.lower for w in t]), noi ci siamo permessi di omettere le parentesi quadre e abbiamo scritto solo: set(w.lower() for w in t) . (Guarda la discussione su "generatore di espressioni" nella Sezione 4.2 per più informazioni al riguardo)

Generando testo random con Bigrams

Possiamo usare un *conditional frequency distribution* per creare una tabella di *bigrams* (coppia di parole). (Abbiamo introdotto i *bigrams* nella sezione 1.3). La funzione bigrams () prende una lista di parole e costruisce una lista di coppie di parole consecutive:

Nell'esempio 2.5 abbiamo lavorato con ogni parola come condizione, e per ognuna di quella abbiamo creato effettivamente una distribuzione di frequenza delle parole seguenti. La funzione generate_model () contiene un semplice loop per generare il testo. Possiamo chiamare la funzione, se scegliamo una parola (come 'living') come il nostro contesto iniziale, dopo, una volta all'interno del loop, possiamo stampare i valori correnti della variabile word, e riportare word ad essere l'elemento più probabile in questo contesto (usando max ()); la prossima volta, attraverso il loop, possiamo usare quella parola nel nostro nuovo contesto. Come potete vedere osservando l'output, questo semplice approccio alla generazione di testi ci porta a rimanere bloccati nei loop; un altro metodo sarebbe quello di scegliere a caso la parola successiva tra quelle disponibili.

```
def generate_model(cfdist, word, num=15):
    for i in range(num):
        print word,
        word = cfdist[word].max()

text = nltk.corpus.genesis.words('english-kjv.txt')
bigrams = nltk.bigrams(text)
cfd = nltk.ConditionalFreqDist(bigrams)
```

```
>>> print cfd['living']
<FreqDist: 'creature': 7, 'thing': 4, 'substance': 2, ',': 1, '.': 1, 'soul': 1>
>>> generate_model(cfd, 'living')
living creature that he said , and the land of the land of the land
```

Esempio 2.5 (code_random_text.py): Figura 2.5: Generating Random Text: questo programma ottiene tutti i bigrams dal testo del libro della Genesi, quindi costruisce un *conditional frequency distribution* per registrare quali parole hanno più probabilità di seguire una parola data; ad esempio, dopo la parola *living*, la parola che più spesso segue è *creature*; la funzione generate_model () usa questi tipi di dati per generare testi random.

La *conditional frequency distribution* è utile per la struttura di dati di molti lavori NLP. I loro metodi maggiormente usati sono riassunti nella Tabella 2.4.

Tabella 2.4:

Conditional frequency distribution di NLTK: i metodi maggiormente usati e le espressioni più usate per definire, accedere e visualizzare la *conditional frequency distribution* di contatori.

Example	Description
`cfdist = ConditionalFreqDist(pairs)`	create a conditional frequency distribution from a list of pairs
`cfdist.conditions()`	alphabetically sorted list of conditions
`cfdist[condition]`	the frequency distribution for this condition
`cfdist[condition][sample]`	frequency for the given sample for this condition
`cfdist.tabulate()`	tabulate the conditional frequency distribution
`cfdist.tabulate(samples, conditions)`	tabulation limited to the specified samples and conditions
`cfdist.plot()`	graphical plot of the conditional frequency distribution
`cfdist.plot(samples, conditions)`	graphical plot limited to the specified samples and conditions
`cfdist1 < cfdist2`	test if samples in `cfdist1` occur less frequently than in `cfdist2`

2.3 Ancora Python: Riutilizzare codici

Fin'ora avrete scritto e riscritto un sacco di codice nell'interprete interattivo Python. Se si fa confusione quando ricopiate un esempio complesso è necessario immetterlo di nuovo. Utilizzare i tasti freccia per accedere e modificare i comandi precedenti è utile, ma può non bastare. In questa sezione vedremo due importanti modi per riutilizzare il codice: i text editors e le funzioni Python.

Creare programmi con un Text Editor

L'interprete interattivo Python esegue le vostre istruzioni non appena le avete digitate. Spesso è meglio scrivere un programma multilinea usando un text editor, poi chiedere a Python di eseguire tutto il programma insieme. Usando IDLE potete fare ciò andando al menu FILE e aprendo una nuova finestra. Provate ora, e immettete il seguente programma su una singola riga:

```
print 'Monty Python'
```

Salvate il programma in un file chiamato monty.py poi andate sul menu RUN e selezionate il comando RUN MODULE. (Impareremo quali moduli sono più brevi) Il risultato nella finestra dell'IDLE principale dovrebbe essere circa così:

```
>>> ================================ RESTART ================================
>>>
Monty Python
>>>
```

Potete anche scrivere from monty import * e vi darà la stessa cosa.

D'ora in poi, avete la possibilità di scegliere se usare l'interprete interattivo o un text editor per creare i vostri programmi. È spesso utile fare un test delle vostre idee usando l'interprete, facendo una revisione di una riga di codice fino a quando non fa quello che ci si aspetta. Una volta pronto, potete incollare il codice all'interno del text editor e continuare ad espanderlo, per alla fine salvare il programma in un file in modo tale da non doverlo scrivere da capo. Dategli un nome breve ma descrittivo, usando tutte lettere minuscole e separando le parole con un trattino basso, usare l'estensione .py per il nome del file, ad esempio monty_python.py.

> **Nota**
>
> **Importante:** i nostri esempi di codice includono i prompt >>> e … come se interagissero direttamente con l'interprete. Man mano che si fanno più complicati, dovreste invece scriversi nell'editor, senza il prompt, e avviarlo dall'editor come è stato fatto sopra. Quando procuriamo dei programmi più lunghi in questo libro lasceremo fuori i prompt per farvi vedere come scriverli all'interno di un file invece di usarli nell'interprete. Potete già vederlo nell'Esempio 2.5 sopra. Fate attenzione che include ancora un paio di linee con il prompt di Python; questa è la parte interattiva del lavoro in cui si controllano alcuni dati e si richiama una funzione.
>
> Ricordate che tutti gli esempi di codici come l'Esempio 2.5 sono scaricabili da http://nltk.org/.

Funzioni

Supponiamo che il vostro lavoro voglia analizzare il testo e per farlo includa diverse forme della stessa parola, e che parte del vostro programma abbia bisogno di elaborare i plurali di una parola singolare. Supponiamo che dobbiate fare questo lavoro in due momenti diversi, uno in cui elaboriamo dei testi, l'altra sull'imput dell'utente.

Anziché ripetere lo stesso codice più volte, è più efficiente localizzare questo lavoro all'interno di una **funzione** (**function**). Una funzione è il nome dato a un blocco di codice che effettua un lavoro ben definito, come abbiamo visto nella sezione 1.1. Una funzione è normalmente definita per prendere alcuni input, usando variabili speciali chiamate parametri, e potrebbe produrre un risultato che chiamiamo **return value** (valore di ritorno). Definiamo una funzione usando la keyword def seguita dal nome della funzione e ogni parametro di input, seguito dal corpo della funzione. Ecco la funzione che abbiamo visto nella Sezione 1.1 (includendo l' che porta le divisioni funzionando come ci aspettavamo):

```
>>> from __future__ import division
>>> def lexical_diversity(text):
...     return len(text) / len(set(text))
```

Abbiamo usato la keyword return per indicare il valore che è prodotto come output della funzione. Nell'esempio sopra, tutto il lavoro della funzione è dato in una dichiarazione di return. Qui abbiamo una definizione equivalente che fa lo stesso lavoro usando più linee di codice. Cambieremo il nome del parametro da text a my_text_data per ricordarvi che è una scelta arbitraria:

```
>>> def lexical_diversity(my_text_data):
...     word_count = len(my_text_data)
...     vocab_size = len(set(my_text_data))
...     diversity_score = word_count / vocab_size
...     return diversity_score
```

Notiamo che abbiamo creato nuove variabile dentro il corpo della funzione. Queste **local variables** (variabili locali) non sono accessibili fuori dalla funzione. Così ora abbiamo definito una funzione con il nome lexical_diversity. Ma definirlo non produce nessun output! Le funzione non fanno nulla finchè non sono "chiamate" (o "invocate"):

Torniamo allo scenario precedente, e definisce appunto una semplice funzione per elaborare i plurali in inglese. La funzione plural () nell'esempio 2.6 prende un nome singolare e ne genera la forma plurale, anche se non è sempre corretto (discuteremo dopo funzioni di maggiore lunghezza nella Sezione 4.4)

```
def plural(word):
    if word.endswith('y'):
        return word[:-1] + 'ies'
    elif word[-1] in 'sx' or word[-2:] in ['sh', 'ch']:
        return word + 'es'
    elif word.endswith('an'):
        return word[:-2] + 'en'
    else:
        return word + 's'
```

```
>>> plural('fairy')
'fairies'
>>> plural('woman')
'women'
```

Esempio 2.6 (code_plural.py): Figura 2.6: Una funzione Python: questa funzione cerca di elaborare il plurale di ogni parola inglese; la keyword def (definire) è seguita dal nome della funzione, dopo un parametro tra parentesi, e due punti; il corpo della funzione è chiamato blocco di codice, cerca di riconoscere i pattern all'interno del lavoro ed elabora la parola di conseguenza; ad esempio se la parola finisce con y, elimina la y e aggiunge *ies*.

La funzione endswith () è sempre associata a una stringa (ad esempio nell'esempio 2.6). Per richiamare funzioni del genere, diamo loro il nome dell'oggetto, un tempo, e dopo il nome della funzione. Queste funzioni sono normalmente conosciute come **methods**.

Moduli

Nel tempo troverete che avrete creato una varietà di utili piccole funzioni di elaborazione di testo, e finirete per copiarli da vecchi programmi a nuovi. Quale file contiene l'ultima versione della funzione che volete usare? Rende la vita molto più semplice se potete raggruppare il vostro lavoro in un singolo posto, e avere accesso alle funzioni precedentemente definite senza fare delle copie.

Per farlo, salvata la vostra funzione (le vostre funzioni) in un file chiamato textproc.py . Ora potete avere accesso al vostro lavoro in modo semplice importantolo dal file:

```
>>> from textproc import plural
>>> plural('wish')
wishes
>>> plural('fan')
fen
```

La precedente funzione per i plurali ha un errore dal momento che il plurale di *fan* è *fans*. Invece di scrivere una nuova versione della funzione, possiamo semplicemente editare quella esistente. Così in ogni

momento, c'è una sola versione della funzione per il plurale e nesusna confuzione su quale debba essere usata.

Una collezione di variabili e definizioni di funzione in un file è chiamata Python **module**. Una collezione di moduli collegati è chiamata un **package** (pacchetto). Il codice NLTK per elaborare il Brown Corpus è un esempio di modulo, e la sua collezione di codice per elaborare tutti i diversi corpora è un esempio di pacchetto. NLTK stesso è un gruppo di pacchetti, chiamati avvolte **library** (libreria).

State attenti!

Se state creando un file che contiene alcune parti del vostro codice Python, non chiamatelo `nltk.py` : potrebbe essere importato al posto del "vero" pacchetto NLTK. Quando importa i moduli, Python prima guarda nella directory usata.

2.4 Risorse Lessicali

Un *lexicon* , o risorsa lessicale, è una collezione di parole e/o frasi con informazioni associate come parti di un discorso o con definizioni sul significato. Le risorse lessicali sono testi secondari e sono creati normalmente e arricchiti con l'aiuto di testi. Ad esempio se abbiamo definito un testo `my_text` , allora costruisce il vocabolario di `vocab=sorted(set(my_text))` ,mentre conta la frequenza di ogni parola nel testo. Entrami `vocab` e `word_freq` sono semplici risorse lessicali. In modo simile una concordanza come quella che abbiamo visto nella Sezione 1.1 ci dà l'informazione sull'uso della parola che potrebbe aiutarci nella preparazione di un dizionario. Una terminologia standard per i lexicons è illustrata nella Figura 2.7. Un **lexical entry** consiste di una **headword** (conosciuta anche come **lemma**) con una informazione addizionale come una parte del discorso o una definizione del senso. Due diverse parole che hanno lo stesso spelling sono chiamate **homonyms**.

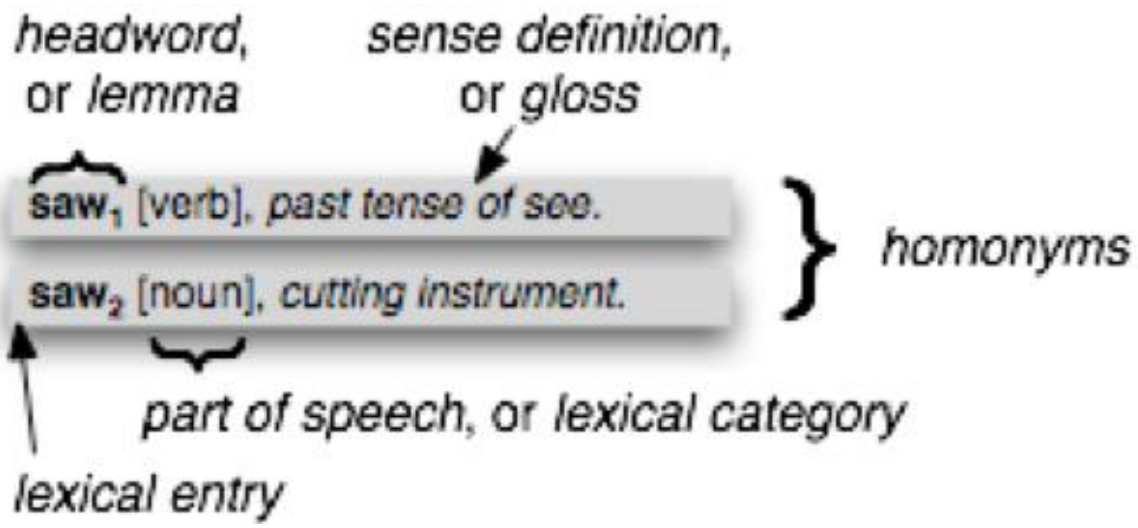


Figura 2.7: Lexicon terminology: le voci lessicali per due lemmi aventi lo stesso spelling (homonyms), forniscono parti del discorso e informazioni

Il tipo più semplice di lexicon non è altro che una lista di parole. Lexicon sofisticati includono strutture complesse con voci individuali che si incrociano. In questa sezione vedremo alcune risorse lessicali incluse in NLTK.

Wordlist Corpora

NLTK include alcuni corpora che non sono altro che una lista di parole (wordlist). Il Words Corpus è il file /urs/share/dict/words di Unix, usato da alcuni correttori ortografici. Possiamo cercare parole inusuali o mis-spelt in un copus testuale, come mostrato nell'esempio 2.8.

```
def unusual_words(text):
    text_vocab = set(w.lower() for w in text if w.isalpha())
    english_vocab = set(w.lower() for w in nltk.corpus.words.words())
    unusual = text_vocab.difference(english_vocab)
    return sorted(unusual)
```

```
>>> unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))
['abbeyland', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary',
'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham',
'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]
>>> unusual_words(nltk.corpus.nps_chat.words())
['aaaaaaaaaaaaaaaa', 'aaahhhh', 'abou', 'abourted', 'abs', 'ack', 'acros',
'actualy', 'adduser', 'addy', 'adoted', 'adreniline', 'ae', 'afe', 'affari', 'afk',
'agaibn', 'agurlwithbigguns', 'ahah', 'ahahah', 'ahahh', 'ahahha', 'ahem', 'ahh', ...]
```

Esempio 2.8 (code_unusual.py): Figura 2.8: Filtraggio di un testo: questo programma calcola il vocabolario di un testo, rimuove tutti gli elementi che ci sono in una wordlist esistente, lasciando solo quelli non comuni o le parole scritte male.

C'è anche un corpus di **stopword**, ovvero, parole d'alta frequenza come *the*, *to* e anche *also* che spesso vogliono filtrare un documento prima di elaborarlo. Stopwords hanno un piccolo contenuto lessicale, e la loro presenza in un file non riesce a distinguerle da altri testi.

```
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['a', "a's", 'able', 'about', 'above', 'according', 'accordingly', 'across',
'actually', 'after', 'afterwards', 'again', 'against', "ain't", 'all', 'allow',
'allows', 'almost', 'alone', 'along', 'already', 'also', 'although', 'always', ...]
```

Definiamo una funzione per elaborare quale frazione di parole in un testo NON è nella lista stopword:

```
>>> def content_fraction(text):
...     stopwords = nltk.corpus.stopwords.words('english')
...     content = [w for w in text if w.lower() not in stopwords]
...     return len(content) / len(text)
...
>>> content_fraction(nltk.corpus.reuters.words())
0.65997695393285261
```

Così, con l'aiuto delle stopwords abbiamo fatto un filtro delle parole nel testo. Si faccia attenzione al fatto che abbiamo combinato due tipi diversi di corpus, usando una risorsa lessicale per filtrare il contenuto di un testo

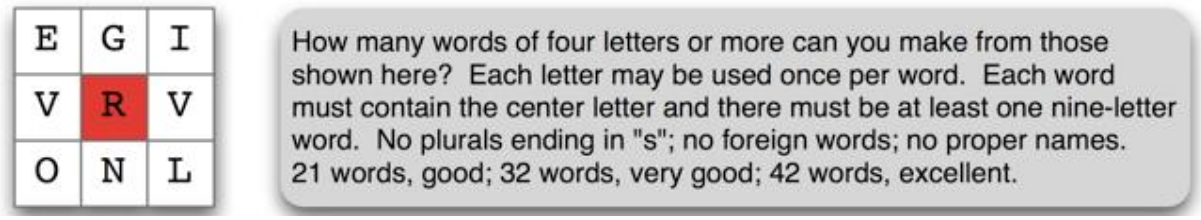


Figura 2.9: Un puzzle di parole: una griglia di lettere scelte a random con regole per creare parole fuori dalle lettere, questo puzzle è conosciuto come "Target".

Una wordlist è utile per risolvere un puzzle di parole, come quello della figura 2.9. Il nostro programma itera attraverso ogni parola e, per ciascuna, controlla quale incontra le condizione. È facile controllare le lettere obbligatorie e i vincoli di lunghezza (e abbiamo cercato solo le parole con sei o più lettere qui). È più complicato controllare che le soluzioni usino solo combinazioni delle lettere che abbiamo dato, specialmente da quando alcune delle lettere date appaiono due volte (qui la lettera v). Il metodo di paragone FreqDist ci permette di verificare che la frequenza di ciascuna lettera nel lavoro preso sia minore o uguale alla frequenza della corrispondente lettera nel puzzle.

```
>>> puzzle_letters = nltk.FreqDist('egivrvonl')
>>> obligatory = 'r'
>>> wordlist = nltk.corpus.words.words()
>>> [w for w in wordlist if len(w) >= 6
...                      and obligatory in w
...                      and nltk.FreqDist(w) <= puzzle_letters]
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

Un'altra wordlist è la Names corpus, che contiene 8 mila nomi categorizzati in base al gender. I nomi maschili e femminili sono salvati in file separati. Vediamo quali nomi appaiono in entrami i file, ad esempio i nomi che cha hanno un gender ambiguo.

```
>>> names = nltk.corpus.names
>>> names.fileids()
['female.txt', 'male.txt']
>>> male_names = names.words('male.txt')
>>> female_names = names.words('female.txt')
>>> [w for w in male_names if w in 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', ...]
```

Si sa che i nomi che finiscono con la lettera *a* sono il più delle volte femminili. Possiamo vedere ciò e altri pattern nel grafico in Figura 2.10, prodotto dal codice seguente. Ricordiamo che name[-1] è l'ultima lettera di name.

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

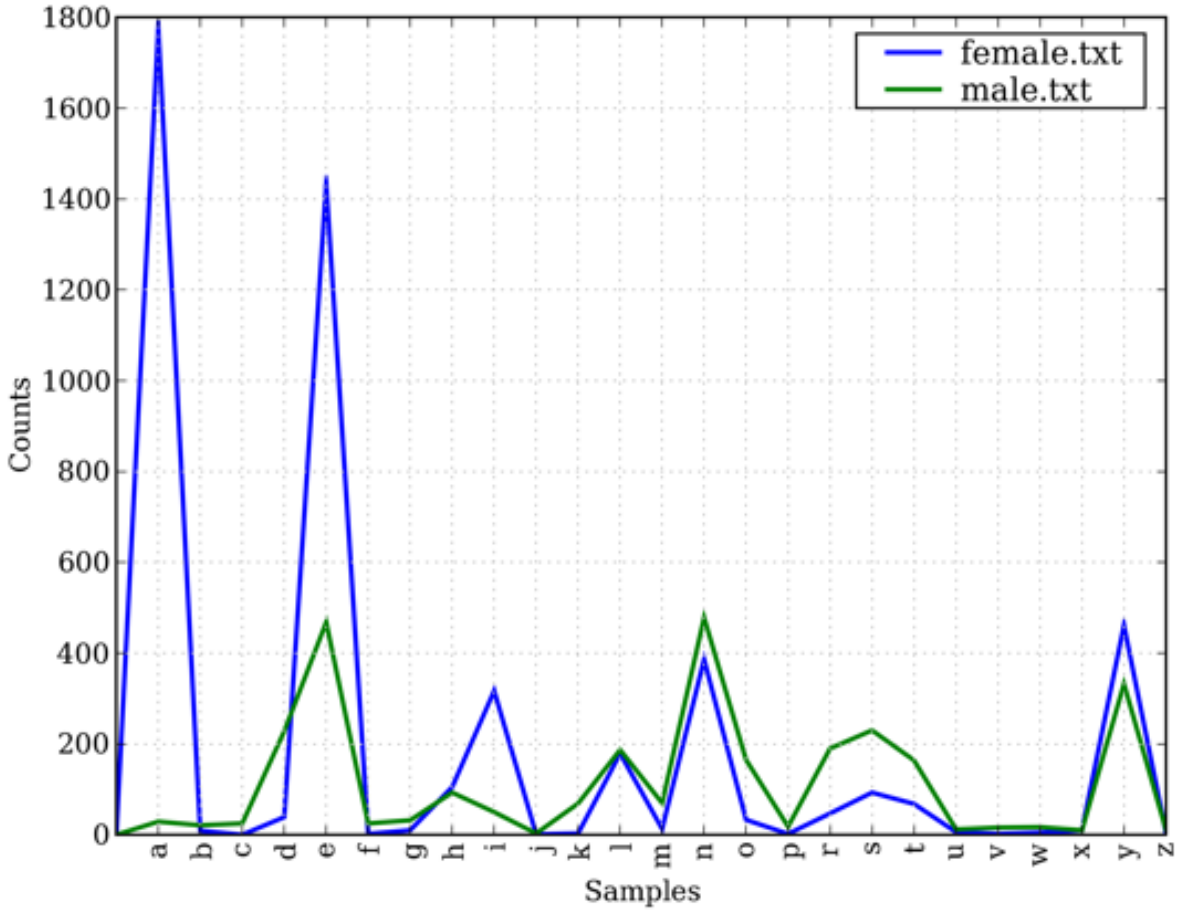


Figura 2.10: *conditional frequency distribution*: questo grafico ci mostra il numero dei nomi femminili e maschili che finiscono con ciascuna lettera dell'alfabeto; la maggior parte dei nomi che finiscono con *a*, *e* o *i* sono femminili; i nomi che finiscono in *h*, *l* sono tanto femminili come maschili, i nomi che finiscono in *k* ,*o* ,*r* ,s e *t*, sono maschili.

A Pronouncig Dictionary

Un tipo un po' più ricco di risorsa lessicale è una tabella (o spreadsheet), che contiene paroli con alcune proprietà per ogni rigo. NLTK include il CMU Pronouncing Dictionary for US English, che è stato elaborato per essere usato da sintetizzatori di discorsi.

```
>>> entries = nltk.corpus.cmudict.entries()
>>> len(entries)
127012
>>> for entry in entries[39943:39951]:
...     print entry
...
('fir', ['F', 'ER1'])
('fire', ['F', 'AY1', 'ER0'])
('fire', ['F', 'AY1', 'R'])
('firearm', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M'])
('firearm', ['F', 'AY1', 'R', 'AA2', 'R', 'M'])
('firearms', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M', 'Z'])
('firearms', ['F', 'AY1', 'R', 'AA2', 'R', 'M', 'Z'])
('fireball', ['F', 'AY1', 'ER0', 'B', 'AO2', 'L'])
```

Per ciascuna parola, questo lexicon elabora una lista di codici fonetici – diverse etichette per ogni suono contrastivo – conosciuto come *phones*. Osservate che *fire* ha due pronunce (in inglese americano): un'unica sillaba F AY1 R , e a due sillabe F AY1 ER0 . I simboli nel CMU Pronouncin Dictionary sono dall'*Arpabet,* descritto dettagliatamente su http://en.wikipedia.org/wiki.Arpabet

Ciascun valore consiste in due parti, possiamo elaborarli individualmente usando una versione più complessa della dichiarazione *for*. Invece di scrivere for entry in entries: , sostituiamo entry con *due* nomi variabili word , pron . Ora, ogni volta nel ciclo, word è assegnata alla prima parte del valore e pron alla seconda parte:

```
>>> for word, pron in entries:
...     if len(pron) == 3:
...         ph1, ph2, ph3 = pron
...         if ph1 == 'P' and ph3 == 'T':
...             print word, ph2,
...
pait EY1 pat AE1 pate EY1 patt AE1 peart ER1 peat IY1 peet IY1 peete IY1 pert ER1
pet EH1 pete IY1 pett EH1 piet IY1 piette IY1 pit IH1 pitt IH1 pot AA1 pote OW1
pott AA1 pout AW1 puett UW1 purt ER1 put UH1 putt AH1
```

Il programma di sopra fa uno scan delle lexicon cercando per ciascun valore la cui pronuncia consiste in tre phones. Se la condizione risulta vera, assegna il contenuto pron alle tre variabili ph1, ph2, e ph3 . Fate attenzione alla forma inusuale della dichiarazione che fa questo lavoro.

Qui c'è un altro esempio della dichiarazione *for*, questa volta usata all'interno di una lista di comprensione. Questo programma trova tutte le parole la cui pronuncia finisce con una sillaba che suona come nicks. Potete usare questo metodo per trovare le parole in rima.

```
>>> syllable = ['N', 'IH0', 'K', 'S']
>>> [word for word, pron in entries if pron[-4:] == syllable]
["atlantic's", 'audiotronics', 'avionics', 'beatniks', 'calisthenics', 'centronics',
'chetniks', "clinic's", 'clinics', 'conics', 'cynics', 'diasonics', "dominic's",
'ebonics', 'electronics', "electronics'", 'endotronics', "endotronics'", 'enix', ...]
```

Fate attenzione che la pronuncia può essere fatta in diversi modi: *nics*, *niks*, *nix* e anche *ntic's* con una *t* silenziosa, per la parola *atlantic's*. Vediamo qualche altro disadattamento tra la pronuncia e la scrittura. Potete riassumere gli obiettivi dei seguenti esempi e spiegare come lavorano?

```
>>> [w for w, pron in entries if pron[-1] == 'M' and w[-1] == 'n']
['autumn', 'column', 'condemn', 'damn', 'goddamn', 'hymn', 'solemn']
>>> sorted(set(w[:2] for w, pron in entries if pron[0] == 'N' and w[0] != 'n'))
['gn', 'kn', 'mn', 'pn']
```

I phones contengono cifre che rappresentano accenti primari (1), secondari (2) e nessun accento (0). Il nostro ultimo esempio vuole definire una funzione per estrarre gli accenti e fare uno scan del lexicon con trovare parole che hanno pattern di accenti particolari.

```
>>> def stress(pron):
...     return [char for phone in pron for char in phone if char.isdigit()]
>>> [w for w, pron in entries if stress(pron) == ['0', '1', '0', '2', '0']]
['abbreviated', 'abbreviating', 'accelerated', 'accelerating', 'accelerator',
'accentuated', 'accentuating', 'accommodated', 'accommodating', 'accommodative',
'accumulated', 'accumulating', 'accumulative', 'accumulator', 'accumulators', ...]
>>> [w for w, pron in entries if stress(pron) == ['0', '2', '0', '1', '0']]
['abbreviation', 'abbreviations', 'abomination', 'abortifacient', 'abortifacients',
'academicians', 'accommodation', 'accommodations', 'accreditation', 'accreditations',
'accumulation', 'accumulations', 'acetylcholine', 'acetylcholine', 'adjudication', ...]
```

Nota

Una sottigliezza del programma è che la nostra funzione stress definita dall'utente è richiamata all'interno della condizione della lista di comprensione. C'è anche un doppiamente annidato loop di *for*. Accadono molte cose qui e potreste voler tornare qui una volta che abbiate acquisito maggiore esperienza nell'uso delle liste di comprensione.

Possiamo usare un *conditional frequency distribution* per aiutarci a trovare dei contrastanti minimi di gruppi di parole. Qui troviamo tutte le parole con la *p-* che consistono in tre suoni, e raggrupparle secondo il loro primo e ultimo suono

```
>>> p3 = [(pron[0]+'-'+pron[2], word)
...       for (word, pron) in entries
...       if pron[0] == 'P' and len(pron) == 3]
>>> cfd = nltk.ConditionalFreqDist(p3)
>>> for template in cfd.conditions():
...     if len(cfd[template]) > 10:
...         words = cfd[template].keys()
...         wordlist = ' '.join(words)
...         print template, wordlist[:70] + "..."
...
P-CH perch puche poche peach petsche poach pietsch putsch pautsch piche pet...
P-K pik peek pic pique paque polk perc poke perk pac pock poch purk pak pa...
P-L pil poehl pille pehl pol pall pohl pahl paul perl pale paille perle po...
P-N paine payne pon pain pin pawn pinn pun pine paign pen pyne pane penn p...
P-P pap paap pipp paup pape pup pep poop pop pipe paape popp pip peep pope...
P-R paar poor par poore pear pare pour peer pore parr por pair porr pier...
P-S pearse piece posts pasts peace perce pos pers pace puss pesce pass pur...
P-T pot puett pit pete putt pat purt pet peart pott pett pait pert pote pa...
P-Z pays p.s pao's pais paws p.'s pas pez paz pei's pose poise peas paiz p...
```

Piuttosto che iterare su tutto il dizionario, possiamo anche accedere, cercando le parole particolari. Useremo la struttura di dati del dizionario di Python, che studieremo attentamente nella Sezione 5.3. cercheremo un dizionario specificando il suo nome, seguito da un **key** (come la parola 'fire') all'interno di parentesi quadre.

```
>>> prondict = nltk.corpus.cmudict.dict()
>>> prondict['fire']
[['F', 'AY1', 'ER0'], ['F', 'AY1', 'R']]
>>> prondict['blog']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'blog'
>>> prondict['blog'] = [['B', 'L', 'AA1', 'G']]
>>> prondict['blog']
[['B', 'L', 'AA1', 'G']]
```

Se proviamo a cercare una parola non esistente avremo keyError. Questo è simile a ciò che succedeva quando facevamo un index con un intero così grande, producendo un IndexError . La parola *blog* manca dal dizionario di pronuncia, così modifichiamo la nostra versione assegnando un valore per questa parola (questo non ha effetti nel corpus NLTK, la prossima volta che accederemo, la parola ancora non si troverà).

Possiamo usare ogni risorsa lessicale per elaborare un testo, ad esempio per filtrare le parole con proprietà lessicali (come i nomi), o mappare ogni singola parola di un testo. Per esempio, la seguente funzione del testo-da-dire cerca ogni singola parola del testo nel dizionario di pronuncia.

```
>>> text = ['natural', 'language', 'processing']
>>> [ph for w in text for ph in prondict[w][0]]
['N', 'AE1', 'CH', 'ER0', 'AH0', 'L', 'L', 'AE1', 'NG', 'G', 'W', 'AH0', 'JH',
'P', 'R', 'AA1', 'S', 'EH0', 'S', 'IH0', 'NG']
```

Wordlist comparativi

Altro esempio di tabelle di lexicon è la **comparative wordlist**. NLTK include quello che viene chiamato **Swadesh worlists**, una lista di circa 200 parole comuni in diverse lingue. Le lingue sono identificate usando un codice di due lettere ISO 369.

```
>>> from nltk.corpus import swadesh
>>> swadesh.fileids()
['be', 'bg', 'bs', 'ca', 'cs', 'cu', 'de', 'en', 'es', 'fr', 'hr', 'it', 'la', 'mk',
'nl', 'pl', 'pt', 'ro', 'ru', 'sk', 'sl', 'sr', 'sw', 'uk']
>>> swadesh.words('en')
['I', 'you (singular), thou', 'he', 'we', 'you (plural)', 'they', 'this', 'that',
'here', 'there', 'who', 'what', 'where', 'when', 'how', 'not', 'all', 'many', 'some',
'few', 'other', 'one', 'two', 'three', 'four', 'five', 'big', 'long', 'wide', ...]
```

Possiamo avere accesso alle parole affini in multiple lingue usando il metodo entries () , specificando una lista di lingue. Con un altro step possiamo convertire questo in un semplice dizionario (vedremo meglio dict () nella Sezione 5.3).

```
>>> fr2en = swadesh.entries(['fr', 'en'])
>>> fr2en
[('je', 'I'), ('tu, vous', 'you (singular), thou'), ('il', 'he'), ...]
>>> translate = dict(fr2en)
>>> translate['chien']
'dog'
>>> translate['jeter']
'throw'
```

Possiamo rendere il nostro semplice traduttore più efficiente aggiungendo altre lingue. Prendiamo il Tedesco-Inglese e lo Spagnolo-Inglese , convertiamole ciascuna in un dizionario usando dict () , facciamo un del nostro originale dizionario translate con queste mappature addizionali:

```
>>> de2en = swadesh.entries(['de', 'en'])    # German-English
>>> es2en = swadesh.entries(['es', 'en'])    # Spanish-English
>>> translate.update(dict(de2en))
>>> translate.update(dict(es2en))
>>> translate['Hund']
'dog'
>>> translate['perro']
'dog'
```

Possiamo paragonare parole tanto nelle lingue germaniche quanto in quelle romane:

```
>>> languages = ['en', 'de', 'nl', 'es', 'fr', 'pt', 'la']
>>> for i in [139, 140, 141, 142]:
...     print swadesh.entries(languages)[i]
...
('say', 'sagen', 'zeggen', 'decir', 'dire', 'dizer', 'dicere')
('sing', 'singen', 'zingen', 'cantar', 'chanter', 'cantar', 'canere')
('play', 'spielen', 'spelen', 'jugar', 'jouer', 'jogar, brincar', 'ludere')
('float', 'schweben', 'zweven', 'flotar', 'flotter', 'flutuar, boiar', 'fluctuare')
```

Lezicons Shoebox e Toolbox

Forse il singolo tool più popolare usato dai linguisti per lavorare i dati è *toolbox*, precedentemente conosciuto come *Shoebox* da quando ha sostituito il campo dei tradizionali linguisti shoebox pieni di file. Toolbox è scaricabile gratuitamente da http://www.sil.org/computing/toolbox/.

Un file toolbox consiste in una raccolta di valori, dove ogni valore è dato da più campi- La maggior parte dei campi è opzionale o ripetibile, ciò vuol dire che questo tipo di risorsa lessicale non può essere trattata come una tabella o uno spreadsheet.

Ecco il dizionario per la lingua Rotokas. Veiamo giusto i primi valori, per la parola *kaa* che significa "imbavagliare":

```
>>> from nltk.corpus import toolbox
>>> toolbox.entries('rotokas.dic')
[('kaa', [('ps', 'V'), ('pt', 'A'), ('ge', 'gag'), ('tkp', 'nek i pas'),
('dcsv', 'true'), ('vx', '1'), ('sc', '???'), ('dt', '29/Oct/2005'),
('ex', 'Apoka ira kaaroi aioa-ia reoreopaoro.'),
('xp', 'Kaikai i pas long nek bilong Apoka bikos em i kaikai na toktok.'),
('xe', 'Apoka is gagging from food while talking.')]), ...]
```

I valori consistono in una serie di coppie di attributi-valori, come ('ps', 'V') per indicare che una parte del discorso è 'V' (verbo), e ('ge', 'gag') per indicare che in inglese è 'gag'. Le ultime tre coppie contengono un esempio di frasi in Rotokas e la traduzione in Tok Pisin e Inglese.

La perdita di struttura dei file toolbox ci rende difficile fare molto di più con loro in questa sede. L'XML fornisce un potente modo di elaborare questo tipo di corpus e ritorneremo su questo argomento nel Capitolo 11.

Nota

La lingua Rotakas è parlata nelle isole di Boungainville, Papua Nuova Guinea. Questo lexicon è stato portato a NLTK da Stuart Robinson. Rotokas è conosciuto per avere un inventario di solo 12 fonemi (suoni contrastanti), http://en.wikipedia.org/wiki/Rotokas_language

2.5 WordNet

WordNet è semanticamente orientato al dizionario inglese, in modo analogo al tradizionale thesaurus, ma con una struttura più ricca. NLTK include l'English WordNet, con 155.287 parole e 117.659 sinonimi. Inizieremo guardando ai sinonimi e come possiamo averne accesso in WordNet.

Senso e sinonimi

Considerando la frase in (1a). Se sostituiamo la parola *autovettura* in (1a) con *automobile*, per avere (1b), il significato della parola rimane più o meno lo stesso.

(1)

a) A Benz è attribuita l'invenzione della autovettura.
b) A Benz è attribuita l'invenzione dell'automobile.

Dato che tutto il resto della frase è rimasto invariato, possiamo concludere che le due parole abbiano lo stesso significato, sono quindi **sinonimi**. Possiamo esplorare queste parole con l'aiuto di WordNet:

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

Quindi *motorcar* (autovettura) ha solo un possibile significato ed è identificato come car.n.01, il primo nome con il significato di *car* (macchina). L'entità di car.n.01 è chiamata un **synset**, o "synonym set", una collezione di sinonimi:

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

Ogni parola di un synset può avere diversi significati, ad esempio *car* può significare anche il trasporto del treno, una gondola, o anche un ascensore auto. Comunque, noi siamo interessati al singolo significato che è comune a tutte le parole del synset più sopra. I Synset possono avere anche una definizione in prosa con 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']
```

Sebbene le definizioni aiutino gli esseri umani a capire il significato che si intende per un synset, le *parole* del synset sono spesso più utili per i nostri programmi. Per eliminare l'ambiguità, vogliamo identificare queste parole come car.n.01.automobile , car.n.01.motorcar ,e così via. Questa coppia di synset con una parola è chiamato lemma. Possiamo prendere tutti i lemmi per un singolo synset, guardiamo a un lemma in particolare, prendiamo il synset corrispondente al lemma, e prendiamo il "nome" del lemma:

```
>>> wn.synset('car.n.01').lemmas
[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')
Lemma('car.n.01.automobile')
>>> wn.lemma('car.n.01.automobile').synset
Synset('car.n.01')
>>> wn.lemma('car.n.01.automobile').name
'automobile'
```

A differenza delle parole *automobile* e *motorcar*, che non sono ambigue e hanno un synset, la parola *car* è ambigua, ha 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']
```

Per convenienza, possiamo accedere a tutti i lemmi che includono la parola *car* 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')]
```

Nota

Sta a voi: Scrivete tutti i significati della parola *dish* a cui potete pensare. Ora cercate la parola con WordNet, usando le stesse operazioni che abbiamo fatto sopra.

La gerarchia di WordNet

I synset di WordNet corrispondono a concetti astratti, e non hanno sempre dei corrispondenti in inglese. Questi concetti sono collegati tra loro tramite una gerarchia. Alcuni concetti sono veramente generici come Entità, Stato, Evento – questi sono chaimati **unique beginners** o root synsets. Altri, come *gas guzzler* o *hatchback*, sono molto più specifici. Una piccola porzione del concetto di gerarchia è illustrato nella Figura 2.11.

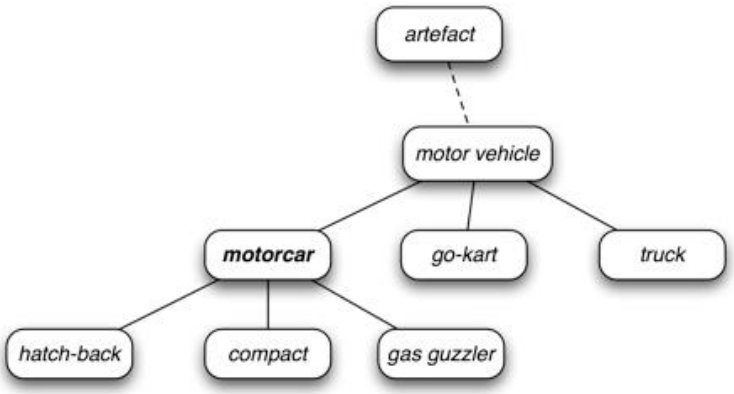


Figura 2.11: Frammento del concetto di gerarchia del WordNet: i nodi corrispondono ai synset; i bordi indicano le relazioni hypernym/hyponym, come la relazione tra concetti subordinati o sovraordinati.

WordNet rende più semplice la navigazione tra i concetti. Ad esempio con un concetto come *motorcar* possiamo vedere concetti più specifici, gli **hyponyms**.

```
>>> motorcar = wn.synset('car.n.01')
>>> types_of_motorcar = motorcar.hyponyms()
>>> types_of_motorcar[26]
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']
```

Possiamo anche navigare sulla gerarchia vedendo gli hypernyms. Alcune parole hanno percorsi multipli perché possono essere classificate in più modi. Ci sono due percorsi per car.n.01 e entity.n.01 perché wheeled_vehicle.n.01 può essere classificata sia come veicolo che come container.

```
>>> 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']
```

Possiamo prendere hypernyms (o root hypernyms) più generici di un synset come segue:

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

Nota

Sta a voi: cercate un grafico di NLTK comodo sul browser WordNet: nltk.app.wordnet(). Esplorate la gerarchia WordNet seguendo i link hypernym e hyponym.

Altre relazioni lessicali

Gli hypernyms e gli hyponyms sono chiamati **relazioni lessicali** perché mettono in relazione un synset con un altro. Queste due relazioni ci permettono di navigare su e giù la gerarchia. Altro modo importante per navigare il network WordNet è dagli elementi ai loro componenti (**meronyms**) o alle cose che contengono (**holonyms**). Per esempio, le parti di un *tree* (albero) sono *trunk*, *crown* (tronco, corona) e così via; le part_meronyms (). La *sostanza* del *tree* è fatto di *heartwood* e *sapwood*, la substance_meronyms(). Una collezione di *trees* forma una *forest* (foresta), il member_holonyms ().

```
>>> wn.synset('tree.n.01').part_meronyms()
[Synset('burl.n.02'), Synset('crown.n.07'), Synset('stump.n.01'),
Synset('trunk.n.01'), Synset('limb.n.02')]
>>> 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 come possono diventare complicate le cose, considerate la parola *mint*, che ha diverse significati molto simili. Possiamo vedere che mint.n.04 fa parte di mint.n.02 e la sostanza di cui è fatta è mint.n.05.

```
>>> 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')]
```

Ci sono anche le relazioni tra i verbi. Ad esempio, il fatto di camminare (*walking*) include il fatto di fare dei passi (*stepping*). Così camminare comporta (**entails**) il fare dei passi. Alcuni verbi hanno multipli derivazioni:

```
>>> wn.synset('walk.v.01').entailments()
[Synset('step.v.01')]
>>> wn.synset('eat.v.01').entailments()
[Synset('swallow.v.01'), Synset('chew.v.01')]
>>> wn.synset('tease.v.03').entailments()
[Synset('arouse.v.07'), Synset('disappoint.v.01')]
```

alcune relazioni lessicali sono date tra i lemmi, ad esempio **antonymy:**

```
>>> wn.lemma('supply.n.02.supply').antonyms()
[Lemma('demand.n.02.demand')]
>>> wn.lemma('rush.v.01.rush').antonyms()
[Lemma('linger.v.04.linger')]
>>> wn.lemma('horizontal.a.01.horizontal').antonyms()
[Lemma('vertical.a.01.vertical'), Lemma('inclined.a.02.inclined')]
>>> wn.lemma('staccato.r.01.staccato').antonyms()
[Lemma('legato.r.01.legato')]
```

Potete vedere le relazioni lessicali, e altri metodi definini su un synset, usando dir () , per esempio: dir(wn.synset('harmony.n.02')) .

Somiglianze semantiche

Abbiamo visto che i synset sono collegati da un complicato network di relazioni lessicali. Dato un particolare synset, possiamo attraversare il network di WordNet per cercare synset con i loro significati correlati. Conoscendo quali parole sono semanticamente collegate è utile per l'idex una collezione di testi, in modo tale da cercare un termine generico come *vehicle* (veicolo) che potrà riscontrare documenti che contengono specifici valori come *limousine*.

Ricordate che ogni singolo synset ha una o più strade di hypernym che lo collegano a un root hypernym come entity.n.01. Due synset collegati attraverso lo stesso root può avere molti hypernyms in comune (Figura 2.11). Se due synset condividono lo stesso specifico hyernym – uno che è più in basso nella gerarchia di hyernym – devono essere strettamente correlati.

```
>>> 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')]
```

Sicuramente sappiamo che *whale* è un termine molto specifico (e *baleen whale* lo è ancor più), mentre *vertebreate* è molto generico e *entity* lo è totalmente. Possiamo quantificare questo concetto di generalità guardando alla 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
```

Misure di similarità sono state definite sulla riscossione di synset di WordNet che incorporano l'intuizione di cui sopra. Ad esempio, path_similarity assegna un valore dell'intervallo 0-1 basato sul più breve percorso che collega un concetto nella gerarchia hypernym (-1 è il risultato nel caso in cui il percorso non venga trovato). Paragonando un synset con se stesso il valore di ritorno sarà 1. Considerando i seguenti risultati si somiglianza, si relazioni *right whale* a *minke whale*, *orca*, *tortoise*, e *novel*. Sebbene i numeri non vogliano dire molto, la loro decrescita è in base a come ci muoviamo nello spazio semantico delle creature marine verso gli oggetti inanimati.

```
>>> right.path_similarity(minke)
0.25
>>> right.path_similarity(orca)
0.16666666666666666
>>> right.path_similarity(tortoise)
0.07692307692307692
>>> right.path_similarity(novel)
0.043478260869565216
```

Nota

Sono disponibili molti altri misurazioni si somiglianza; potete scrivere help (wn) per trovare più informazioni. NLTK include anche VerbNEt, sempre gerarchico e collegato a WordNet. Vi si può accedere con nltk.corpus.verbnet.

2.6 Sommario

- Un corpo di testo è un'ampia, strutturata raccolta di testi. NLTK necessita di molti corpi di testo, ad es. il Corpo Brown, nltk.corpus.brown.
- Alcuni corpi di testo sono categorizzati, ad es. per genere o argomento; a volte, le categorie di un corpo si sovrappongono l'una con l'altra.

- Una distribuzione condizionata di frequenza è una raccolta di distribuzioni di frequenza, ognuna per una condizione differente. Tali distribuzioni possono essere utilizzare per il conteggio della frequenza delle parole, dato un contesto o un genere.
- I programmi Python, più che quelli lunghi poche righe, dovrebbero essere inseriti usando un compilatore di testo, salvati in un file con formato.py e lanciati utilizzando un'istruzione import.
- Le funzioni Python permettono di associare un nome a un particolare codice di blocco e di riutilizzare quel codice ogni volta che necessita .
- Alcune funzioni, conosciute come "metodi", sono associate ad un oggetto e viene assegnato il nome dell'oggetto, seguito da un periodo seguito da una funzione, come questo: x.funct(y), ad es., word.isalpha().
- Per To find out about some variable v, type help(v) in the Python interactive interpreter to read the help entry for this kind of object.
- WordNet è un dizionario di inglese semanticamente-orientato, che consiste di una serie di sinonimi (*synsets)* e organizzato in una rete.
- Alcune funzioni non sono disponibili di default, ma devono essere lanciate utilizzando l'istruzione di Python import.

2.7 Ulteriori letture

Materiali extra per questo capitolo sono caricati all'indirizzo http://www.nltk.org/, che include links a risorse web disponibili gratuitamente. I metodi di corpo sono riassunti nel Corpo HOWTO, all'indirizzo http://www.nltk.org/howto, e documentati in modo esteso nella documentazione online.

Fonti significative di corpi pubblicati sono il *Linguistic Data Consortium* (LDC) e l' *European Language Resources Agency* (ELRA). Centinaia di corpi testi e dialoghi annotati sono disponibili in dozzine di linguaggi. Licenze non commerciali permettono di utilizzare i dati per l'insegnamento e la ricerca. Per alcuno corpi, licenze commerciali sono disponibili (ma ad un prezzo maggiore)

Queste e molte altre risorse linguistiche sono state documentate usando Metadata OLAC, e possono essere ricercate tramite l' homepage OLAC all'indirizzo http://www.language-archives.org/. *Corpora List* è una mailing list per discussioni riguardanti i corpi e si possono trovare risorse cercando cercando nella lista di archivi . L'inventario più completo delle lingue presenti al mondo è *Ethnologue*, http://www.ethnologue.com/. Di 7,000 lingue, solo poche dozzine hanno sostanziali risorse digitali adatte all'utilizzo in NLP.

Questo capitolo ha toccato l'ambito del **Corpo Linguistico**. Altri libri utili in quest'area includono (Biber, Conrad, & Reppen, 1998), (McEnery, 2006), (Meyer, 2002), (Sampson & McCarthy, 2005), (Scott & Tribble, 2006). Ulteriori letture sull' analisi quantitativa di dati in linguistica sono: (Baayen, 2008), (Gries, 2009), (Woods, Fletcher, & Hughes, 1986).

La descrizione originale di WordNet è (Fellbaum, 1998). Sebbene WordNet sia stato sviluppato originariamente per la ricerca nel campo della psicolinguistica, è attualmente ampiamente utilizzato in *NLP and Information Retrieval*. Altri WordNets stanno per essere sviluppati per molte altre lingue, come documentato all'indirizzo, http://www.globalwordnet.org/. Per uno studio di misure di *similarity WordNet*, vedere (Budanitsky & Hirst, 2006).

Altri temi toccata in questo capitolo sono stati la fonetica e la semantica lessicale, e si rimandano I lettori ai capitoli 7 e 20 di (Jurafsky & Martin, 2008).

2.8 Exercises

1. ☼ Creare una variabile phrase che contenga una lista di parole. Provare le operazioni descritte in questo capitolo, incluse addizioni, moltiplicazioni, indicizzazioni, sezionamento e classificazione
2. ☼ Utilizzare il modulo di corpo per esplorare austen-persuasion.txt. Quanti *word tokens* ha questo libro? Quanti *word types*?
3. ☼ Utilizzare il lettore del corpus Brown nltk.corpus.brown.words() o il lettore del corpo di testo Web nltk.corpus.webtext.words() per accedere ad alcuni testi campione in due differenti generi.
4. ☼ Leggere nei testi degli indirizzi *State of the Union*, utilizzando il lettore di corpo state_union. Contare le ricorrenza di men, women, e people in ogni documento. Cos'è accaduto all'uso di queste parole nel tempo?
5. ☼ Indagare le relazioni olonimo-meronimo per alcuni nomi. Ricordare che vi sono tre tipo di relazione olonimo-meronimo, bisogna così utilizzare: member_meronyms(), part_meronyms(),substance_meronyms(), member_holonyms(), part_holonyms(), e substance_holonyms().
6. ☼ Nella discussione sulle wordlists comparative, si era creato un oggetto chiamato translate,che poteva essere ricercato usando parole sia in italiano che in tedesco, in modo tale da ottenere parole corrispondenti in inglese. Che problema potrebbe sorgere con questo approccio? Si può suggerire un modo per evitare questo problema?
7. ☼ Secondo Strunk and White's *Elements of Style*, la parola *however*, usata all'inizio di una frase, significa "in whatever way" o "to whatever extent", e non "nevertheless". Gli autori propongono questo esempio di utilizzo corretto: *However you advise him, he will probably do as he thinks best.* (http://www.bartleby.com/141/strunk3.html) Utilizzare lo strumento di concordanza per studiare l'attuale uso di questa parola nei vari testi considerati. Vedere anche il *LanguageLog* che "posta" "Fossilized prejudices about 'however'" all'indirizzo http://itre.cis.upenn.edu/~myl/languagelog/archives/001913.html
8. ◑ Definire una distribuzione condizionata di frequenza sul corpo Names, che permette di verificare quali lettere *initial* sono più frequenti per males vs. females (cf. Figura 2.10).
9. ◑ Prendere un paio di testi e studiare le differenza tra loro, in termini di vocabolario, ricchezza di vocaboli, genere, etc. Si può trovare un paio di parole che abbia significati abbastanza differenti nei due testi, come *monstrous* in *Moby Dick* e in *Sense and Sensibility*?
10. ◑ Leggere l'articolo della BBC: *UK's Vicky Pollards 'left behind'* http://news.bbc.co.uk/1/hi/education/6173441.stm. L'articolo dà la seguente statistica sul linguaggio dei teenager: "le 20 parole più usate, incluse yeah, no, but e like, contano come circa un terzo di tutte le parole." Quanti *word types* contano come un terzo di tutti i *word tokens*, per la varietà delle fonti del testo? Cosa si può concludere riguardo questa statistica? Leggere ulteriore material a riguardo su *LanguageLog*, all'indirizzo http://itre.cis.upenn.edu/~myl/languagelog/archives/003993.html.
11. ◑ Indagare la tavola delle distribuzioni modali e cercare altri schemi. Cercare di spiegarli in termini della propria comprensione impressionistica dei differenti generi. Si possono trovare altre classi chiuse di parole che mostrano differenze significative lungo i diversi generi?
12. ◑ Il CMU Pronouncing Dictionary contiene pronounce multiple di certe parole. Quante parole distinte contiene? Quale frazione di parole in questo dizionario ha più di una possibile pronuncia?
13. ◑ Che percentuale di nomi *synsets* non ha iponomi? Si possono ottenere tutti i nomi *synsets* usando wn.all_synsets('n').
14. ◑ Definire una funzione supergloss(s)che prende un *synset* s come suo argomento e rimanda una stringa che consiste della concatenazione della definizione di s, e delle definizioni di tutti gli ipernomi e iponomi di s.
15. ◑ Scrivere un programma per trovare tutte le parole che ricorrono almeno tre volte in corpo Corpus.

16. ◑ Scrivere un programma per generare una tavola di *lexical diversity scores* (es. token/type ratios), come visto in Table 1.1. Includere il pieno set dei generi del corpo Brown (nltk.corpus.brown.categories()). Che genere ha la più bassa *diversity* (il più alto numero di tokens per tipo)? È ciò che ci si aspetta?
17. ◑ Scrivere una funzione che trovi le 50 parole di un testo che ricorrono più frequentemente e che non siano stopwords.
18. ◑ Scrivere un programma per stampare I 50 bigrammi (coppie di parole adiacenti)più frequenti di un testo, omettendo i bigrammi che contengono stopwords.
19. ◑ Scrivere un programma per creare una tabella di frequenze di parole per genere, come quella data in _sec-extracting-text-from-corpora for modals. Scegliere delle parole e cercare di trovare parole la cui presenza (o assenza) sia tipica di un genere. Discutere i risultati.
20. ◑ Scrivere una funzione word_freq() che prende una parola e il nome di una sezione del corpo Brown come argomenti, e computa la frequenza della parola in quella parola del corpo.
21. ◑ Scrivere un programma per calcolare il numero di sillabe contentue in un testo, facendo uso del CMU Pronouncing Dictionary.
22. ◑ Definire una funzione hedge(text) che processi un testo e produca una nuova versione con la parola 'like' tra ogni terza parola .
23. ★ **La legge di Zipf**: porre *f(w)* come la frequenza di una parola *w* in un testo *free*. Supporre che tutte le parole di un testo siano ordinate secondo la loro frequenza, con al primo posto quella più frequente. La legge di Zipf afferma che la frequenza di una tipologia di parola è inversamente proporzionale al suo ordine(es. $f \times r = k$, per alcune costanti k). Per esempio, il 50mo tipo di parola più comune dovrebbe ricorrere tre volte tanto frequentemente quanto il 150mo tipo.
    1. Write a function to process a large text and plot word frequency against word rank using pylab.plot. Do you confirm Zipf's law? (Hint: it helps to use a logarithmic scale). What is going on at the extreme ends of the plotted line? [Scrivere una funzione che processi la frequenza di un ampio testo e schema contro l'ordine delle parole usando pylab.plot. La legge di Zipf viene confermata? (Accenno: aiutarsi utilizzando una scala logaritmica). Cosa succede agli estremi della riga tracciata?]
    2. Generare un testo casuale, es, usando random.choice("abcdefg "), preoccupandosi di includere il carattere "spazio". Per prima cosa ci sarà bisogno di import random. Usare l'operatore di concatenazione della stringa per accumulare caratteri in una stringa (molto) lunga. Poi *"tokenize"* questa stringa, e generare lo schema Zipf come in precedenza, e comparare i due schemi. Cosa si può dire sulla legge di Zipf alla luce di questo?
24. ★ Modificare il programma di generazione di testo in Example 2.5 , per svolgere i seguenti compiti:
    1. Memorizzare le *n* parole più probabili in una lista words, poi scegliere casualmente una parola dalla lista usando random.choice(). (Per prima cosa sarà necessario import random.)
    2. Selezionare un genere particolare, come una sezione del corpo Brown, o una genesis translation, uno dei testi di Gutenberg, o uno dei testi Web. Preparare il modello su questo corpo e realizzarlo per generare un testo random. Si dovrebbe sperimentare con differenti start words. Quanto intelligibile è il testo? Discutere la forza e debolezza di questo metodo di generazione casual del testo.
    3. Ora preparare il sistema usando due generi distinti e sperimentare con un testo che produce con il genere ibrido. Discutere le proprie osservazioni.
25. ★ Definire una funzione find_language() che prenda una stringa come suo argomento e restituisca una lista di linguaggi che hanno quella stringa come parola. Utilizzare il corpo udhr e limitare le ricerche ai files nella codifica Latin-1.
26. ★ Qual è il fattore diramante della gerarchia dell'ipernome? Ad es. Per ogni *noun synset* che ha iponome — o children nella gerarchia di ipernome — quanti ne hanno in media? È possibile ottenere tutti i *noun synsets* usando wn.all_synsets('n').
27. ★ Il polisemo di una parola è il numero di significati che possiede. Usando WordNet, si può determinare che il *dog* ha 7 significati grazie a: len(wn.synsets('dog', 'n')). Computare la polisemia media di nomi, verbi,aggettivi e avverbi secondo WordNet.

28. ⋆ Utilizzare una delle misure predefinite di *similarity* per realizzare la *similarity* di ognuna delle seguenti coppie di parole. Ordinare le coppie per *similarity* decrescente. Quanto si avvicina la tua classificazione all'ordine dato di seguito, un ordine stabilito sperimentalmente da (Miller & Charles, 1998): car-automobile, gem-jewel, journey-voyage, boy-lad, coast-shore, asylum-madhouse, magician-wizard, midday-noon, furnace-stove, food-fruit, bird-cock, bird-crane, tool-implement, brother-monk, lad-brother, crane-implement, journey-car, monk-oracle, cemetery-woodland, food-rooster, coast-hill, forest-graveyard, shore-woodland, monk-slave, coast-forest, lad-wizard, chord-smile, glass-magician, rooster-voyage, noon-string.

Su questo documento...

Questo documento è Revision: 8464 Mon 14 Dec 2009 10:58:42 EST

3 Elaborazione di testi semplici

La più grande risorsa di testi è sicuramente il Web. È utile avere a disposizione una raccolta di testi esistenti da esplorare, come ad esempio i corpora che abbiamo visto nei capitoli precedenti. Comunque, probabilmente avrete le vostre fonti di testi già ben presenti e necessitate di imparare come avere accesso ad esse.

L'obiettivo di questo capitolo è di rispondere alle seguenti domande:

1. Come possiamo elaborare programmi per accedere ai testi dai file locali e dal web, al fine di entrare in possesso di una illimitata gamma di materiale linguistico?
2. Come possiamo suddividere i documenti in singole parole e simboli di punteggiatura, in modo che possiamo attuare gli stessi tipi di analisi che abbiamo fatto con i corpora di testi nei capitoli precedenti?
3. Come possiamo elaborare programmi per produrre un output formattato e salvarlo in un file?

Per indirizzare queste domande, affronteremo concetti chiave in NLP, inclusi la "tokenizzazione" e i suoi derivati. Lungo il percorso consoliderete la vostra conoscenza di Python e imparerete circa le sequenze, i file, e le espressioni regolari. Dato che la maggior parte del testo presente sul web è in formato HTML, vedremo anche come possiamo fare a meno della marcatura .

Note

Importante: Da questo capitolo in avanti, i nostri campioni di programmi presuppongono una sessione interattiva da parte vostra con le seguenti asserzioni istruttorie

```
>>> from __future__ import division
>>> import nltk, re, pprint
```

3.1 Accedere al testo dal web e dal disco

Libri elettronici

Un piccolo campione di testi dal Progetto Guntenberg è presente nella raccolta corpus NLTK. Comunque, potreste essere interessati ad analizzare altri testi provenienti dal Progetto Gutenberg. Potete esplorare il catalogo di 25,000 libri online sul sito http://www.gutenberg.org/catalog/, e ottenere un URL per un file di testo ASCII. Sebbene il 90% dei testi presenti nel Progetto Gunteberg sia in inglese, esso include materiali in più di 50 lingue diverse, compreso il Catalano, il Cinese, il Danese, il Finlandese, il Francese, il Tedesco, l'Italiano, il Portoghese e lo Spagnolo (con più di 100 testi per ogni lingua).

Il testo numero 2554 è una traduzione in Inglese di *Orgoglio e Pregiudizio,* e possiamo averne accesso nel seguente modo.

```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by
Fyodor Dostoevsky\r\n'
```

Nota

Il comando `read( )` impiegherà qualche secondo affinché scarichi un libro così grande. Se state usando un proxy di internet che non è correttamente individuato da Python, probabilmente dovrete specificare il proxy manualmente così come segue:

```
>>> proxies = {'http': 'http://www.someproxy.com:3128'}
>>> raw = urlopen(url, proxies=proxies).read()
```

La variabile `raw` contiene una stringa con 1,176,831 caratteri. (Possiamo notare che questa è una stringa, che usa `type ( raw)`.) Questo è il semplice contenuto del libro, che include molti dettagli che non ci interessano come spazi bianchi, interruzioni di riga e righe vuote. Da notare le `/r` e `/n` nella linea d'apertura del file, che indicano come Python visualizza il ritorno a capo e il carattere di nuova riga (il file deve essere generato da un sistema operativo Windows). Per l'elaborazione del nostro linguaggio, vogliamo scomporre la stringa in parole e segni di punteggiatura, come abbiamo visto nel Capitolo 1. Questo passaggio è chiamato tokenizzazione, e genera la nostra struttura familiare, una lista di parole e segni di punteggiatura.

```
>>> tokens = nltk.word_tokenize(raw)
>>> type(tokens)
<type 'list'>
>>> len(tokens)
255809
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and',
'Punishment', ',', 'by']
```

Notare che NLTK è necessario per la tokenizzazione, ma non per ogni altra precedente funzione di apertura di un URL e di lettura di questo in una stringa. Se adesso compissimo il passo successivo di creare un testo NLTK da questa lista, effettueremo tutte le altre procedure linguistiche che abbiamo visto nel Capitolo 1, con l'ordinaria lista di operazioni come dividere:

```
>>> text = nltk.Text(tokens)
>>> type(text)
<type 'nltk.text.Text'>
>>> text[1020:1060]
['CHAPTER', 'I', 'On', 'an', 'exceptionally', 'hot', 'evening', 'early',
'in',
'July', 'a', 'young', 'man', 'came', 'out', 'of', 'the', 'garret', 'in',
'which', 'he', 'lodged', 'in', 'S', '.', 'Place', 'and', 'walked', 'slowly',
',', 'as', 'though', 'in', 'hesitation', ',', 'towards', 'K', '.', 'bridge',
```

```
'.']
>>> text.collocations()
Katerina Ivanovna; Pulcheria Alexandrovna; Avdotya Romanovna; Pyotr
Petrovitch; Project Gutenberg; Marfa Petrovna; Rodion Romanovitch;
Sofya Semyonovna; Nikodim Fomitch; did not; Hay Market; Andrey
Semyonovitch; old woman; Literary Archive; Dmitri Prokofitch; great
deal; United States; Praskovya Pavlovna; Porfiry Petrovitch; ear rings
```

Notare che il *Progetto Gutenberg* compare come collocazione. Ciò accade perché ogni testo scaricato dal Progetto Gutenberg contiene un'intestazione con il nome del testo, l'autore, il nome delle persone che hanno visionato e corretto il testo, una licenza, e via di seguito. A volte questa informazione appare a piè di pagina e alla fine del file. Non possiamo individuare con certezza dove il testo comincia e dove finisce, e per questo dobbiamo ricorrere ad un'ispezione manuale del file, per scoprire stringhe singole che segnano l'inizio e la fine, prima di ridurre `raw` in modo che rappresenti il contenuto e nient'altro:

```
>>> raw.find("PART I")
5303
>>> raw.rfind("End of Project Gutenberg's Crime")
1157681

>>> raw = raw[5303:1157681]
>>> raw.find("PART I")
0
```

I metodi `find ( )` e `rfind ( )` ("reverse find"- "ricerca inversa", n.d.t) ci aiutano ad ottenere i giusti valori dell'indice da usare per dividere la stringa. Noi scriviamo `raw` con questa parte, così adesso comincia con "PART I" e investe (ma non includendo) la frase che marca la fine del contenuto.

Questo è stato il nostro primo contatto con la realtà del web: i testi trovati sul web possono contenere materiale indesiderato, e potrebbe non esserci un modo automatico per rimuoverlo. Ma con uno sforzo in più possiamo estrarre il materiale che ci serve.

Rapporti con HTML

Molti dei testi presenti sul web sono in formato di documento HTML. Potete usare un web browser per salvare una pagina come testo su un file locale, poi accedervi come descritto nella sezione sui file sottostante. Comunque, se siete intenzionati a compiere questa operazione spesso, è più facile usare Python per fare direttamente il lavoro. Il primo passo è lo stesso del precedente, usando `urlopen`. Per divertimento sceglieremo una storia della BBC News chiamata *Blondes to die out in 200 years*, una leggenda metropolitana fatta passare dalla BBC come un fatto scientifico stabilito:

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

Potete scrivere `print html` per vedere il contenuto HTML in tutta la sua gloria, inclusi i meta tags, una image map, gli script JavaScript, i form, e tabelle.

Estrapolare il testo da un documento in formato HTML è un'operazione abbastanza comune tanto che NLTK prevede la funzione di supporto `nltk.clean_html()`, che prende una stringa in HTML e la traduce in testo semplice. Possiamo poi tokenizzare la stringa per avere la nostra familiare struttura testuale:

```
>>> raw = nltk.clean_html(html)
>>> tokens = nltk.word_tokenize(raw)
>>> tokens
['BBC', 'NEWS', '|', 'Health', '|', 'Blondes', "'", 'to', 'die', 'out', ...]
```

Il risultato contiene ancora materiale indesiderato inerente la navigazione del sito ed altro ad essa collegata. Con alcune difficoltà ed errori potrete trovare gli indici di inizio e fine del contenuto e selezionare i segni di interesse, e inizializzare un testo come prima.

```
>>> tokens = tokens[96:399]
>>> text = nltk.Text(tokens)
>>> text.concordance('gene')
 they say too few people now carry the gene for blondes to
last beyond the next tw
t blonde hair is caused by a recessive gene . In order for a
child to have blonde
to have blonde hair , it must have the gene on both sides of
the family in the gra
there is a disadvantage of having that gene or by chance .
They don ' t disappear
ondes would disappear is if having the gene was a disadvantage
and I do not think
```

Nota
Per un'elaborazione più accurata di HTML, usare il pacchetto *Beautiful Soup*, disponibile su `http://www.crummy.com/software/BeautifulSoup/`

Elaborazione risultati dei motori di ricerca

Il web può essere concepito come un vasto corpus di testi non commentati. I motori di ricerca forniscono efficienti mezzi per la ricerca di questa grande quantità di testi destinati ad essere adeguati esempi linguistici. Il vantaggio principale dei motori di ricerca è la gamma: dal momento che state cercando in un vasto insieme di documenti, è molto probabile che troviate ogni modello linguistico a cui siete interessati. Inoltre, è possibile fare uso di modelli molto particolari, che corrisponderebbero a solo uno o due esempi su un campione più piccolo, ma che potrebbero corrispondere a decine di migliaia di esempi se la ricerca è eseguita sul web. Un altro vantaggio dei motori di ricerca è che sono molto semplici da usare. Così, forniscono uno strumento molto valido per verificare velocemente una teoria, per vedere se è ragionevole.

Tabella 3.1

I risultati di Google per Collocazioni: Il numero di questi risultati incluse le parole *absolutely* o *definitely*, seguite dalle parole *adore, love, like,* o *prefer*. (Liberman, in *LanguageLog,* 2005).

Google hits	***adore***	***love***	***like***	***Prefer***
absolutely	289,000	905,000	16,200	644
definitely	1,460	51,000	158,000	62,600

ratio	198:1	18:1	1:10	1:97

Sfortunatamente, i motori di ricerca hanno delle carenze significanti. Per prima cosa, la gamma dei modelli di ricerca accessibili è rigorosamente ristretta. Diversamente dai corpora locali, in cui è possibile scrivere programmi per cercare arbitrariamente modelli complessi, di solito i motori di ricerca permettono di cercare solo per singole parole o stringhe di parole, a volte con metacaratteri. In secondo luogo, i motori di ricerca danno risultati incoerenti, e possono dare risultati molto diversi quando usati in tempi diversi o in aree geografiche diverse. Quando il contenuto è stato duplicato attraverso più siti, i risultati della ricerca possono essere moltiplicati. Infine, la marcatura nei risultati ottenuti da un motore di ricerca può cambiare in modo imprevedibile, rendendo vano ogni metodo basato su modelli di localizzazione di un particolare contenuto (un problema che è stato migliorato grazie all'uso delle API del motore di ricerca).

Nota
Tocca a te: Cerca sul web `"the of"` (nelle virgolette). Basandoci su un conteggio vasto, possiamo concludere che *the of* è una collocazione frequente in inglese?

Elaborazione dei Feed RSS

La blogosfera è un'importante risorsa di testo, in registri sia formali che informali. Con l'aiuto della terza parte della libreria Python chiamata *Universal Feed Parser*,scaricabile gratuitamente dal sito http://feedparser.org/, possiamo avere accesso al contenuto di un blog, come mostrato qui sotto:

```
>>> import feedparser
>>> llog =
feedparser.parse("http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title']
u'Language Log'
>>> len(llog.entries)
15
>>> post = llog.entries[2]
>>> post.title
u"He's My BF"
>>> content = post.content[0].value
>>> content[:70]
u'<p>Today I was chatting with three of our visiting graduate students f'
>>> nltk.word_tokenize(nltk.html_clean(content))
>>> nltk.word_tokenize(nltk.clean_html(llog.entries[2].content[0].value))
[u'Today', u'I', u'was', u'chatting', u'with', u'three', u'of', u'our',
u'visiting',
u'graduate', u'students', u'from', u'the', u'PRC', u'.', u'Thinking',
u'that', u'I',
u'was', u'being', u'au', u'courant', u',', u'I', u'mentioned', u'the',
u'expression',
u'DUI4XIANG4', u'\u5c0d\u8c61', u'("', u'boy', u'/', u'girl', u'friend',
u'"', ...]
```

Notare che le stringhe dei risultati hanno una u come prefisso per indicare che sono delle stringhe Unicode (vedi la Sezione 3.3). Con un po' di lavoro in più, possiamo scrivere programmi per creare un piccolo corpus di messaggi testuali pubblicati sui blog, e usarlo come la base per il nostro lavoro NLP.

Lettura dei File Locali

Al fine di leggere un file locale, dobbiamo usare la funzione integrata di Python `open( )`, seguita dal metodo `read( )`. Supponiamo di avere un file `document.txt`, possiamo caricare i suoi contenuti in questo modo:

```
>>> f = open('document.txt')
>>> raw = f.read()
```

Nota

Tocca a te: create un file chiamato `documenti.txt` usando un editor di testo, scriveteci dentro poche linee di testo e salvatelo come testo normale. Se state usando IDLE, selezionate il comando Nuova Finestra nel menu File, scrivendo il testo richiesto in questa finestra, e poi salvate il file come `document.txt` nella directory che IDLE offre nella finestra di dialogo pop-up. In seguito, nell'interprete Python, aprite il file usando `f = open('document.txt')`, poi controllatene i contenuti usando `print f.read()`.

Molte cose potrebbero andare male quando proverete ad eseguire questa operazione. Se l'interprete non dovesse trovare il vostro file, dovreste visualizzare un errore come questo:

```
>>> f = open('document.txt')
Traceback (most recent call last):
File "<pyshell#7>", line 1, in -toplevel-
f = open('document.txt')
IOError: [Errno 2] No such file or directory: 'document.txt'
```

Per verificare che il file che si sta cercando di caricare è realmente presente nella giusta directory, usate il comando *Open* di IDLE nel menu *File;* ciò farà apparire una lista di tutti i files presenti nella directory in cui IDLE sta operando. Un'alternativa è esaminare la directory in corso dentro Python:

```
>>> import os
>>> os.listdir('.')
```

Un altro problema che potreste incontrare accedendo ad un file di testo consiste nelle regole per la newline, che sono diverse da sistema operativo a sistema operativo . La funzione integrata `open()` ha un secondo parametro per controllare come è stato aperto il file: `open('document.txt', 'rU')` — `'r'` indica l'apertura del file in sola lettura (il default), e `'U'` sta per "Universal", che ci invita ad ignorare le diverse convenzioni usate per segnare le newline.

Posto che il file si possa aprire, ci sono diversi metodi per leggerlo. Il metodo `read()` crea una stringa con i contenuti dell'intero file:

```
>>> f.read()
'Time flies like an arrow.\nFruit flies like a banana.\n'
```

Ricordiamo che i caratteri `'\n'` sono **newline**; questo equivale alla pressione del tasto *Enter* su una tastiera per andare a capo.

Possiamo anche leggere un file una riga alla volta usando un ciclo `for` :

```
>>> f = open('document.txt', 'rU')
>>> for line in f:
...     print line.strip()
Time flies like an arrow.
Fruit flies like a banana.
```

Qui abbiamo usato il metodo `strip()` per rimuovere il carattere newline alla fine della riga di input.

Il corpus di file di NLTK può essere accessibile attraverso questi metodi. Dobbiamo semplicemente usare `nltk.data.find()` per avere il nome del file per ogni elemento del corpus. In seguito possiamo aprire e leggerlo nel modo che abbiamo illustrato sopra:

```
>>> path = nltk.data.find('corpora/gutenberg/melville-
moby_dick.txt')
>>> raw = open(path, 'rU').read()
```

Estrazione dei testi da PDF, MSWord e altri Formati Binari

Il testo ASCII e il testo HTML sono formati leggibili dall'uomo. Il testo a volte ci si presenta in formati binari- come PDF e MSWord- che possono essere aperti solo usando dei software specializzati. Le librerie di terze parti come `pypdf` e `pywin32` forniscono l'accesso a questi formati. L'estrazione del testo da documenti multi-colonna è particolarmente impegnativo. Per una conversione occasionale di pochi documenti, è più semplice aprire il documento con un'applicazione appropriata, poi salvarlo come testo nell'unità locale, e poi accedervi come descritto sopra. Se il documento è ancora presente sul web, potete inserire il suo URL nella casella di ricerca di Google. Il risultato della ricerca spesso include un link ad una versione HTML del documento, che può essere salvato come testo.

Acquisizione di un Input di un utente

Qualche volta vogliamo acquisire il testo che un utente ha immesso quando sta interagendo con il nostro programma. Per sollecitare l'utente a scrivere una linea di input, richiamate la funzione Python `raw_input()`. Dopo aver salvato l'input come variabile, possiamo manipolarlo come abbiamo fatto per le altre stringhe.

```
>>> s = raw_input("Enter some text: ")
Enter some text: On an exceptionally hot evening early in
July
>>> print "You typed", len(nltk.word_tokenize(s)),
"words."
You typed 8 words.
```

La pianificazione NLP

La figura 3.1 riassume ciò che abbiamo esposto in questa sezione, incluso il processo di costruzione di un vocabolario che abbiamo affrontato nel Capitolo1. (Un passaggio, la normalizzazione, verrà affrontato nella Sezione 3.6).

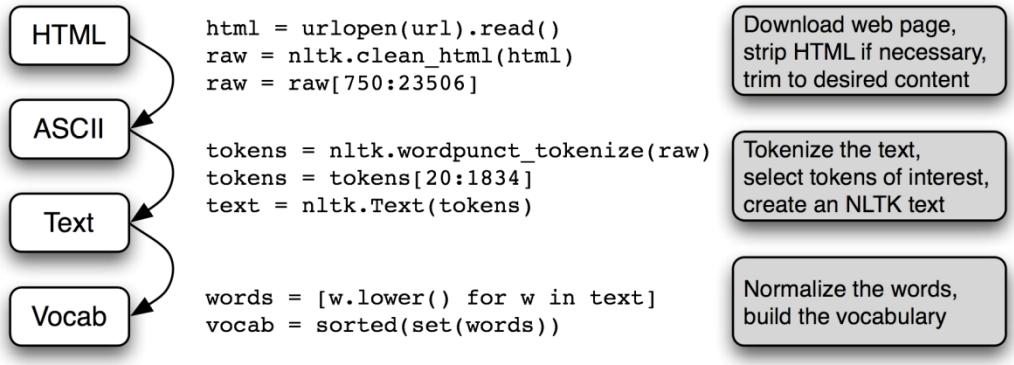


Figura 3.1: La procedura di pianificazione: apriamo un URL e leggiamo il suo contenuto HTML, rimuoviamo la marcatura e selezioniamo una porzione di caratteri; questa verrà poi tokenizzata e eventualmente convertita in un `nltk.Text`, possiamo inoltre rendere in minuscolo tutte le parole e estrarre il vocabolario.

Ci sarebbe tanto da dire su questa procedura di pianificazione. Per capirla bene, ci aiuta essere precisi sul tipo di ogni variabile che menziona. Appuriamo la natura di ogni oggetto Python `x` usando `type(x)`, per esempio `type(1)` è `<int>` dal momento che `1` è un numero intero.

Quando carichiamo i contenuti di un URL o di un file, e rimuoviamo la marcatura HTML, stiamo avendo a che fare con delle stringhe, il dato di dato Python `<str>` (Approfondiremo il concetto di stringhe nella sezione 3.2):

```
>>> raw = open('document.txt').read()
>>> type(raw)
<type 'str'>
```

Quando tokenizziamo una stringa produciamo una lista (di parole), e questa è il tipo di dato Python `<list>`. La normalizzazione e l'ordinamento delle liste produce altre liste:

```
>>> tokens = nltk.word_tokenize(raw)
>>> type(tokens)
<type 'list'>
>>> words = [w.lower() for w in tokens]
>>> type(words)
<type 'list'>
>>> vocab = sorted(set(words))
>>> type(vocab)
<type 'list'>
```

La scrittura di un oggetto determina quali operazioni si possono compiere su di esso. Così, per esempio, possiamo aggiungere a una lista ma non a una stringa:

```
>>> vocab.append('blog')
>>> raw.append('blog')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

Parimenti, possiamo unire stringhe con altre stringhe,e liste con altre liste, ma non possiamo unire stringhe con liste:

```
>>> query = 'Who knows?'
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> query + beatles
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'list' objects
```

3.2 Stringhe: elaborazione dei testi al livello più elementare

È arrivato il momento di studiare un fondamentale dato di scrittura che abbiamo accuratamente evitato finora. Nei capitoli precedenti ci siamo concentrati su un testo come una lista di parole. Non ci siamo soffermati molto sulle parole e su come sono trattate in linguaggio di programmazione. Utilizzando l'interfaccia del corpus di NLTK siamo stati abili nell'ignorare i file da cui questi testi provengono. Il contenuto di una parola, e di un file, sono rappresentati dai linguaggi di programmazione come un fondamentale tipo di dati noto come **stringa**. In questa sezione esploreremo le stringhe nel dettaglio, e mostreremo la connessione tra stringhe, parole, testi e file.

Operazioni di base con le stringhe

Le stringhe vengono specificate usando singole virgolette o doppie virgolette, come specificato sotto. Se una stringa contiene una virgoletta separata, dobbiamo apporre il carattere di fuga backslash alla virgoletta così che Python capisca che il carattere letterale virgoletta è intenzionale, o anche mettere la stringa in doppie virgolette. In caso contrario, la virgoletta nella stringa verrà interpretata come una virgoletta di chiusura della stringa, e l'interprete Python segnalerà un errore di sintassi:

```
>>> monty = 'Monty Python'
>>> monty
'Monty Python'

>>> circus = "Monty Python's Flying Circus"
>>> circus
"Monty Python's Flying Circus"

>>> circus = 'Monty Python\'s Flying Circus'
>>> circus
"Monty Python's Flying Circus"
```

```
>>> circus = 'Monty Python's Flying Circus'
  File "<stdin>", line 1
    circus = 'Monty Python's Flying Circus'
                            ^
SyntaxError: invalid syntax
```

A volte le stringhe si estendono su più righe. Python ci fornisce molti modi per immetterle. Nel prossimo esempio, una sequenza di due stringhe è ridotta a una singola stringa. Abbiamo bisogno di usare un backslash o le parentesi così che l'interprete capisca che la frase non è finita dopo la prima riga.

```
>>> couplet = "Shall I compare thee to a Summer's day?"\

...            "Thou are more lovely and more temperate:"
>>> print couplet
Shall I compare thee to a Summer's day?Thou are more lovely and more
temperate:
>>> couplet = ("Rough winds do shake the darling buds of May,"

...            "And Summer's lease hath all too short a date:")
>>> print couplet
Rough winds do shake the darling buds of May,And Summer's lease hath all too
short a date:
```

Sfortunatamente il metodo sopra riportato non ci dà una nuova riga tra le due righe del sonetto. Invece, possiamo usare una stringa con triple-virgolette come segue:

```
>>> couplet = """Shall I compare thee to a Summer's day?
... Thou are more lovely and more temperate:"""
>>> print couplet
Shall I compare thee to a Summer's day?
Thou are more lovely and more temperate:
>>> couplet = '''Rough winds do shake the darling buds of May,
... And Summer's lease hath all too short a date:'''
>>> print couplet
Rough winds do shake the darling buds of May,
And Summer's lease hath all too short a date:
```

Adesso che possiamo definire le stringhe, possiamo provare alcune semplici operazioni su di esse. Per prima cosa guardiamo alla operazione +, conosciuta come **concatenazione.** Produce una nuova stringa che è una copia delle due originali stringhe copiate insieme finale-con-finale. Notate che la concatenazione non compie niente di eccezionale come inserire uno spazio tra le parole. Possiamo persino moltiplicare le stringhe:

```
>>> 'very' + 'very' + 'very'
'veryveryvery'

>>> 'very' * 3
'veryveryvery'
```

Nota

Tocca a te: provate a far funzionare il seguente codice, poi provate ad usare la vostra conoscenza delle operazioni di stringa + e * per dimostrare come funziona. Fate attenzione a distinguere tra la stringa ' ', che è un unico carattere di spaziatura, e '', che è una stringa vuota.

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1]
>>> b = [' ' * 2 * (7 - i) + 'very' * i for i in a]
>>> for line in b:
...     print line
```

Abbiamo visto che le operazioni di addizione e la moltiplicazione si applicano alle stringhe, non solo ai numeri. Comunque, notate che non possiamo usare la sottrazione o la divisione con le stringhe:

```
>>> 'very' - 'y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> 'very' / 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Questi messaggi di errore sono un altro esempio di come Python ci dice che abbiamo messo i nostri dati in confusione. Nel primo caso, siamo stati avvisati che l'operazione di sottrazione (es., -) non può essere applicata a oggetti di scrittura `str` (stringhe), mentre nel secondo, siamo stati avvisati che la divisione non può accettare `str` e `int` come suoi due operandi.

Stampare le stringhe

Finora, quando abbiamo voluto guardare i contenuti di una variabile o vedere i risultati di un calcolo, abbiamo solo scritto il nome della variabile nell'interprete. Possiamo vedere anche il contenuto di una variabile usando il comando `print`:

```
>>> print monty
Monty Python
```

Notare che non ci sono virgolette questa volta. Quando verifichiamo una variabile scrivendo il suo nome nell'interprete, l'interprete stampa la rappresentazione Python del suo valore. Dal momento che è una stringa, il risultato è virgolettato. Comunque, quando diciamo all'interprete di `print` il contenuto della variabile, non vediamo virgolette dal momento che non ce ne sono nella stringa.

Il comando `print` ci permette di visualizzare più di un elemento in una riga in più modi, come mostrato qui sotto:

```
>>> grail = 'Holy Grail'
>>> print monty + grail
Monty PythonHoly Grail
>>> print monty, grail
Monty Python Holy Grail
>>> print monty, "and the", grail
Monty Python and the Holy Grail
```

Estrarre singoli caratteri

Come abbiamo visto nella Sezione 1.2 per le liste, le stringhe sono indicizzate, partendo da zero. Quando indicizziamo una stringa, prendiamo uno dei suoi caratteri (o lettere). Un singolo carattere non è nulla di speciale- è solo una stringa di lunghezza 1.

```
>>> monty[0]
'M'
>>> monty[3]
't'
>>> monty[5]
' '
```

Come con le liste, se proviamo ad estrapolare un indice che è fuori dalla stringa commettiamo un errore:

```
>>> monty[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Ancora come con le liste, possiamo usare incidi negativi per le stringhe, dove `-1` è l'indice dell'ultimo carattere. Indici positivi e negativi ci danno due modi per riferirci a qualsiasi posizione in una stringa. In questo caso, quando una stringa ha una lunghezza di 12, gli indici 5 e -7 si riferiscono entrambi allo stesso carattere (uno spazio). (Notare che `5 = len(monty) - 7.`)

```
>>> monty[-1]
'n'
>>> monty[5]
' '
>>> monty[-7]
' '
```

Possiamo scrivere il ciclo `for` per iterare il carattere nella stringa. Questo comando `print` finisce con una virgola finale, che è il modo in cui diciamo a Python di non stampare una nuova riga alla fine.

```
>>> sent = 'colorless green ideas sleep furiously'
>>> for char in sent:
...     print char,
...
c o l o r l e s s  g r e e n  i d e a s  s l e e p  f u r
i o u s l y
```

Possiamo conteggiare singoli caratteri. Dobbiamo ignorare la distinzione maiuscolo-minuscolo normalizzando tutto in minuscolo, e filtrando tutti i caratteri non alfabetici:

```
>>> from nltk.corpus import gutenberg
>>> raw = gutenberg.raw('melville-moby_dick.txt')
>>> fdist = nltk.FreqDist(ch.lower() for ch in raw if
ch.isalpha())
>>> fdist.keys()
['e', 't', 'a', 'o', 'n', 'i', 's', 'h', 'r', 'l', 'd', 'u',
'm', 'c', 'w',
'f', 'g', 'p', 'b', 'y', 'v', 'k', 'q', 'j', 'x', 'z']
```

Questo ci dà le lettere dell'alfabeto, con le lettere che compaiono più di frequente per prime (ciò è un po' complicato e lo spiegheremo meglio avanti). Dovreste visualizzare la distribuzione usando `fdist.plot()`. La relativa frequenza di caratteri in un testo può essere usata automaticamente identificando il linguaggio del testo.

Estrarre le sottostringhe

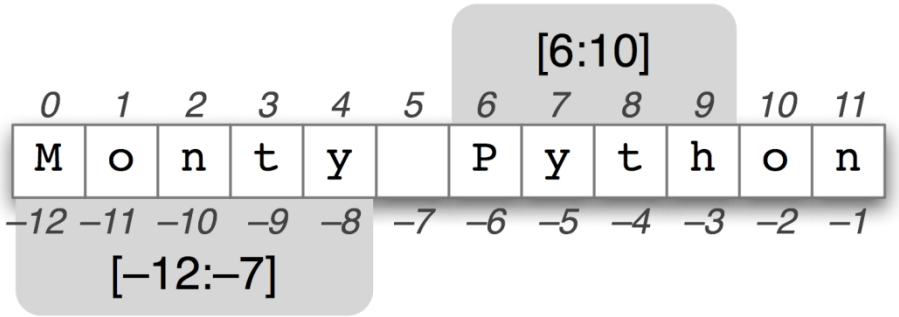


Figura 3.2: Divisione della stringa. La stringa "Monty Python" è mostrata con gli indici sia negativi che positivi; due sottostringhe sono selezionate usando la notazione "slice" (parte, n.d.t). La slice `[m,n]` contiene i caratteri dalla posizione `m` fino alla `n-1`.

Una sottostringa è ogni sezione continua di una stringa che vogliamo estrarre per ulteriori elaborazioni. Possiamo facilmente accedere alle sottostringhe usando la stessa notazione di divisione che abbiamo usato per le liste (vedi la Figura 3.2). Per esempio il codice che segue accede alla sottostringa che comincia all'indice 6, fino (ma non includendo) all'indice 10:

```
>>> monty[6:10]
```

```
'Pyth'
```

Qui vediamo i caratteri `'P'`, `'y'`, `'t'` e `'h'` che corrispondono a `monty[6]` ... `monty[9]` ma non a `monty[10]`. Questo perché una parte *comincia* al primo indice ma finisce ad *uno prima* dell'indice finale.

Possiamo anche dividere con indici negativi- la stessa regola basilare di partire dall'indice iniziale e finire ad uno prima dell'indice finale; qui ci fermiamo prima del carattere di spaziatura.

```
>>> monty[-12:-7]
'Monty'
```

Come con le parti di lista, se omettiamo il primo valore, la sottostringa comincia all'inizio della stringa. Se omettiamo in secondo valore, la sottostringa continua fino alla fine della stringa:

```
>>> monty[:5]
'Monty'
>>> monty[6:]
'Python'
```

Proviamo se una stringa contiene una particolare sottostringa usando l'operatore `in`, come segue:

```
>>> phrase = 'And now for something completely different'
>>> if 'thing' in phrase:
...     print 'found "thing"'
found "thing"
```

Possiamo anche trovare la posizione di una sottostringa all'interno di una stringa, usando `find()`:

```
>>> monty.find('Python')
6
```

Nota
Tocca a te: Elabora una frase ed assegnala ad una variabile, es. sent = 'my sentence...'. Adesso scrivi le espressioni di divisione per estrarre le singole parole. (Questo è ovviamente un modo poco conveniente per elaborare le parole di un testo!)

Ulteriori operazioni sulle stringhe

Python ha un supporto completo per l'elaborazione delle stringhe. Un riassunto, che include alcune operazioni che non abbiamo ancora esaminato, è mostrato nella Tabella 3.2. Per ulteriori informazioni sulle stringhe, scrivere help(str) nel prompt di Python.

Tabella 3.2:

Metodi utili per le stringhe: operazioni sulle stringhe in aggiunta agli esempi mostrate nella Tabella 1.4, tutti i metodi producono una nuova stringa o lista

Method	Functionality
`s.find(t)`	Indice di prima istanza della stringa `t` in `s` (`-1` se non trovato)
`s.rfind(t)`	Indice di ultima istanza `t` in `s` (`-1` se non trovato)
`s.index(t)`	Come `s.find(t)` eccetto se produce `ValueError` quando non trovata
`s.rindex(t)`	Come `s.rfind(t)` eccetto se produce `ValueError` quando non trovata
`s.join(text)`	Combina le parole del testo in una stringa usando `s` come collante
`s.split(t)`	Cambia `s` in una lista dove una `t` è trovata (spazi per impostazione predefinita)
`s.splitlines()`	Cambia `s` in una lista di stringhe, una per riga
`s.lower()`	Una versione in minuscolo della stringa `s`
`s.upper()`	Una versione maiuscola della stringa `s`
`s.title()`	Una versione con tutte le iniziali in maiuscolo della stringa `s`
`s.strip()`	Una copia di `s` senza spazi iniziali o finali
`s.replace(t, u)`	

Le differenze tra liste e stringhe

Stringhe e liste sono entrambe tipi di sequenze. Possiamo separarle indicizzandole o dividendole, e possiamo unirle concatenandole. Comunque, non possiamo unire tra loro stringhe e liste:

```
>>> query = 'Who knows?'
>>> beatles = ['John', 'Paul', 'George', 'Ringo']
>>> query[2]
'o'
>>> beatles[2]
'George'
>>> query[:2]
'Wh'
>>> beatles[:2]
['John', 'Paul']
>>> query + " I don't"
"Who knows? I don't"
>>> beatles + 'Brian'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
>>> beatles + ['Brian']
['John', 'Paul', 'George', 'Ringo', 'Brian']
```

Quando apriamo un file in lettura in un programma Python, otteniamo una stringa che corrisponde al contenuto dell'intero file. Se usiamo il ciclo `for` per elaborare gli elementi di questa stringa, tutto ciò che ne possiamo ricavare sono i caratteri individuali- non possiamo scegliere la granularità. Per contrasto, gli elementi di una lista possono essere tanto grandi quanto piccoli come preferiamo: per esempio, possono essere paragrafi, sentenze, frasi, parole, caratteri. In questo modo le liste hanno il vantaggio che ci permettono di essere flessibili circa gli elementi che contengono, e parimenti essere flessibili circa ogni elaborazione a valle. Di conseguenza, una delle prime cose che abbiamo più piacere di fare in un pezzo di codice NLP è tokenizzare una stringa in una lista di stringhe (Sezione 3.7). Per converso, quando vogliamo scrivere il nostro risultato in un file, o in un terminale, solitamente li formatteremo in una stringa (Sezione 3.9).

Liste e stringhe non hanno esattamente la stessa funzione. Le liste hanno in aggiunta la facoltà di far cambiare i loro elementi:

```
>>> beatles[0] = "John Lennon"
>>> del beatles[-1]
>>> beatles
['John Lennon', 'Paul', 'George']
```

D'altra parte se proviamo a compiere questa operazione con una *stringa*- cambiando il carattere 0 in `query` a `'F'`- abbiamo:

```
>>> query[0] = 'F'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

Ciò accade perché le stringhe sono **immutabili**- non si possono cambiare le stringhe una volta create. Comunque, le liste sono **mutevoli**, e il loro contenuto può essere modificato in qualsiasi momento. Come risultato, le liste supportano le operazione che modificano il valore originale piuttosto che produrre un nuovo valore.

Nota
Tocca a te: consolidate la vostra conoscenza delle stringhe provando alcuni esercizi sulle stringhe alla fine di questo capitolo.

Elaborazione dei testi con Unicode

Il nostro programma avrà spesso a che fare con linguaggi diversi, e diversi insiemi di caratteri. Il concetto di "testo testo" non esiste. Se vivete in un contesto anglofono probabilmente userete ASCII, forse senza rendervene conto. Se vivete in Europa potreste usare uno degli estesi insieme di carattere latini, che contengono caratteri come "ø" per il Danese e il Norvegese, "ő" per l'Ungherese, "ñ" per lo Spagnolo e il Bretone, e "ň" per il Ceco e lo Slovacco. In questa sezione, daremo uno sguardo a come usare Unicode per elaborare quei testi che usano una gamma di caratteri non-ASCII.

Cos'è Unicode?

Unicode supporta oltre un milione di caratteri. Ad gni carattere è assegnato un numero, chiamato **code point**. In Python, i code point sono scritti nella forma `\u`*XXXX, dove XXXX è il numero nella forma di 4 cifre esadecimali.*

Dentro un programma, possiamo manipolare le stringhe Unicode come stringhe normali. Comunque, quando i caratteri Unicode sono memorizzati nei file o visualizzati in un terminale, devono essere codificati come un flusso di byte. Alcune codifiche (come ASCII e Latin-2) usano un singolo byte per code point, così possono supportare solo un piccolo sottoinsieme di Unicode, sufficiente per un singolo linguaggio. Altre codifiche (come UTF-8) usano più byte e possono rappresentare l'intera gamma dei caratteri Unicode.

Il testo nei file sarà in una particolare codificazione, perciò abbiamo bisogno di alcuni meccanismi per tradurlo in Unicode- la traduzione in Unicode è detta ***decoding****. Viceversa, per trascrivere Unicode in un file o un terminale, per prima cosa abbiamo bisogno di tradurlo in una codifica adatta- questa traduzione di Unicode è detta* ***encoding****, ed è illustrata nella Figura 3.3.*

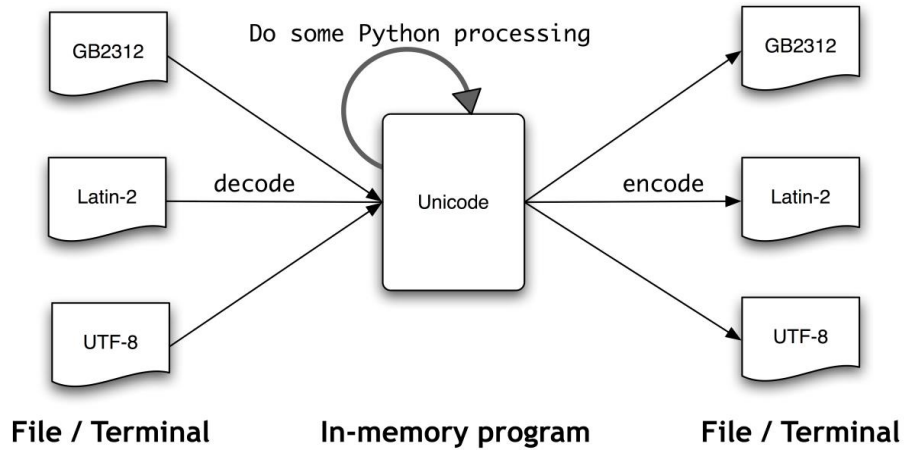


Figura 3.3: Decodifica e codifica di Unicode

Dalla prospettiva Unicode i caratteri sono entità astratte che possono essere realizzati come uno o più **glyph**. Solo i glyph possono apparire sullo schermo o essere stampati su carta. Un font (carattere) è una mappatura da caratteri a glyph.

Estrazione di testi codificati dai file

Supponiamo di avere un piccolo file di testo, e che sappiamo come è codificato. Per esempio, `polish-lat2.txt`, come il nome suggerisce, è un frammento di testo in Polacco (dalla versione polacca di Wikipedia; vedi `http://pl.wikipedia.org/wiki/Biblioteka_Pruska`). Questo file è codificato come Latin-2, conosciuto anche come ISO-8859-2. La funzione `nltk.data.find()` individua il file per noi.

```
>>> path = nltk.data.find('corpora/unicode_samples/polish-
lat2.txt')
```

Il modulo Python `codecs` prevede funzioni per leggere dati codificati in stringhe di Unicode, e per trascrivere stringhe Unicode in forme codificate. La funzione `codecs.open()` prende un parametro codificato per specificare se il codice del file debba essere letto o scritto. Così importiamo il modulo `codecs`, e chiamiamolo con il codice `'latin2'` per aprire il nostro file in Polacco come Unicode.

```
>>> import codecs
>>> f = codecs.open(path, encoding='latin2')
```

Per una lista di parametri di codifica permessi dai `codecs`, vedi http://docs.python.org/lib/standard-econdings.html. Notare che possiamo scrivere dati unicode-codificati in un file usando `f = codecs.open(path, 'w', encoding='utf-8')`.

Il testo letto dall'oggetto file `f` verrà restituito in Unicode. Come abbiamo chiarito precedentemente, al fine di vedere questo testo su un terminale, abbiamo bisogno di codificarlo, usando una codifica adatta. La codifica specifica di Python `unicode_escape` è un codice fittizio che converte tutti i caratteri non-ASCII nelle loro rappresentazioni `\u`*XXXX. Il codice punta sulla gamma ASCII 0-127 ma sotto 256 sono rappresentati nella forma a due cifre* `\x`*XX*.

```
>>> for line in f:
...     line = line.strip()
...     print line.encode('unicode_escape')
Pruska Biblioteka Pa\u0144stwowa. Jej dawne zbiory znane
pod nazw\u0105
"Berlinka" to skarb kultury i sztuki niemieckiej.
Przewiezione przez
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny
\u015al\u0105sk, zosta\u0142y
odnalezione po 1945 r. na terytorium Polski. Trafi\u0142y
do Biblioteki
Jagiello\u0144skiej w Krakowie, obejmuj\u0105 ponad 500
tys. zabytkowych
archiwali\xf3w, m.in. manuskrypty Goethego, Mozarta,
Beethovena, Bacha.
```

La prima riga sopra illustra una stringa escape preceduta dalla stringa escape `/u`, ossia `/u0144`. Il carattere Unicode pertinente apparirà sullo schermo come il glyph ń. Nella terza riga dell'esempio precedente, vediamo `/xf3` , che corrisponde al glyph ó, e si trova all'interno della gamma 128-255.

In Python, una stringa letterale Unicode può essere specificata facendo precedere una semplice stringa letteraria da una `u`, come in `u'hello'`. I caratteri Unicode arbitrari sono definiti usando la sequenza escape `\u`*XXXX nella stringa letterale Unicode. Troviamo il numero intero ordinale di un carattere usando* `ord()`. *Per esempio:*

```
>>> ord('a')
97
```

La notazione esadecimale in quattro cifre per 97 è 0061, così possiamo definire una stringa letterale Unicode con l'appropriata sequenza escape:

```
>>> a = u'\u0061'
>>> a
u'a'
>>> print a
A
```

Notare che il comando Python `print` sta presupponendo un errore di codifica del carattere unicode, cioè ASCII. Comunque, ń è fuori dalla gamma ASCII, perciò non può essere stampato a meno che non specifichiamo un codifica. Nell'esempio che segue, abbiamo specificato che `print` dovrebbe usare il `repr()` della stringa, che restituisce la sequenza escape UTF-8 (della forma /xXX) piuttosto che provare a tradurre i glyph.

```
>>> nacute = u'\u0144'
>>> nacute
u'\u0144'
>>> nacute_utf = nacute.encode('utf8')
>>> print repr(nacute_utf)

'\xc5\x84'
```

Se il vostro sistema operativo e il vostro ambiente sono impostati per restituire i caratteri codificati UTF-8 , dovreste essere capaci di dare il comando Python `print nacute_utf` e vedere ń sul vostro schermo.

Nota

Ci sono molti fattori che determinano ciò che glyph sono visualizzati sullo schermo. Se siete sicuri di avere la codifica corretta, ma il vostro codice Python sbaglia ancora nel produrre il glyph che vi aspettate, dovreste verificare di avere installati i font necessari sul vostro sistema operativo.

Il modulo `unicodedata` ci fa controllare le proprietà dei caratteri Unicode. Nell'esempio che segue, selezioniamo tutti i caratteri nella terza riga del nostro testo in Polacco fuori dalla gamma ASCII e stampiamo i loro valori escape UTF-8, seguiti dai loro codici di numeri integrali usando la convenzione standard Unicode (es, prefissando le cifre esadecimale con `U+`), seguiti dai loro nomi Unicode.

```
>>> import unicodedata
>>> lines = codecs.open(path,
encoding='latin2').readlines()
>>> line = lines[2]
>>> print line.encode('unicode_escape')
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny
\u015al\u0105sk, zosta\u0142y\n
>>> for c in line:
...     if ord(c) > 127:
...         print '%r U+%04x %s' % (c.encode('utf8'),
ord(c), unicodedata.name(c))
'\xc3\xb3' U+00f3 LATIN SMALL LETTER O WITH ACUTE
'\xc5\x9b' U+015b LATIN SMALL LETTER S WITH ACUTE
'\xc5\x9a' U+015a LATIN CAPITAL LETTER S WITH ACUTE
'\xc4\x85' U+0105 LATIN SMALL LETTER A WITH OGONEK
'\xc5\x82' U+0142 LATIN SMALL LETTER L WITH STROKE
```

Se sostituite il `%r` (che restituisce il valore `repr()`) con il `%s` nella stringa di formato dell'esempio di codice riportato sopra, e se il vostro sistema supporta UTF-8, dovreste vedere un output come il seguente:

ó U+00f3 LATIN SMALL LETTER O WITH ACUTE
ś U+015b LATIN SMALL LETTER S WITH ACUTE
Ś U+015a LATIN CAPITAL LETTER S WITH ACUTE
ą U+0105 LATIN SMALL LETTER A WITH OGONEK
ł U+0142 LATIN SMALL LETTER L WITH STROKE

In alternativa, potresti avere bisogno di sostituire la codifica `'utf8'` nell'esempio con `'latin2'`, dipendendo ancora dai dettagli del vostro sistema.

I prossimi esempi illustrano come i metodi stringa di Python e il modulo `re` accettano le stringhe Unicode.

```
>>> line.find(u'zosta\u0142y')
54
>>> line = line.lower()
>>> print line.encode('unicode_escape')
niemc\xf3w pod koniec ii wojny \u015bwiatowej na dolny
\u015bl\u0105sk, zosta\u0142y\n
>>> import re
>>> m = re.search(u'\u015b\w*', line)
>>> m.group()
u'\u015bwiatowej'
```

I tokenizzatori NLTK accettano come input le stringhe Unicode, e viceversa restituiscono stringhe Unicode come output.

```
>>> nltk.word_tokenize(line)
[u'niemc\xf3w', u'pod', u'koniec', u'ii', u'wojny',
u'\u015bwiatowej',
u'na', u'dolny', u'\u015bl\u0105sk', u'zosta\u0142y']
```

Usare la propria codificazione locale in Python

Se siete abituati a lavorare con caratteri in una particolare codificazione locale, probabilmente si desidera essere in grado di utilizzare i metodi standard per l'inserimento e la modifica di stringhe in un file Python. Per fare questo, bisogna includere la stringa `'# -*- coding: <coding> -*-'` come prima o seconda riga del file. Notare che `<coding>` deve essere una stringa come `'latin-1'`, `'big5'` o `'utf-8'` (vedi Figura 3.4).

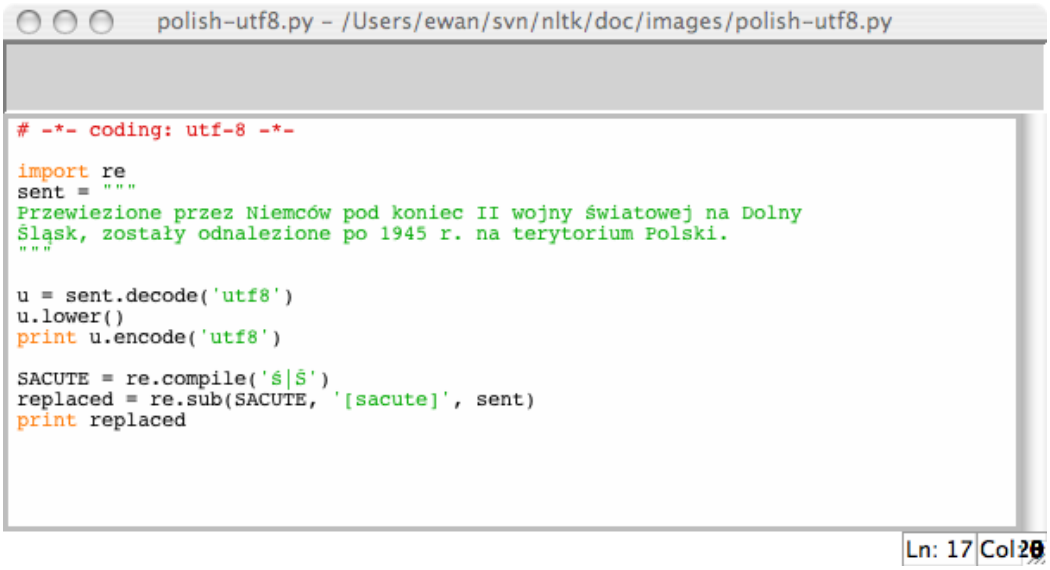


Figura 3.4: Unicode e IDLE: errore della stringa codificata UTF-8 nel revisore IDLE; ciò richiede che un font appropriato sia posto nelle preferenze di IDLE; qui abbiamo scelto Courier CE.

L'esempio superiore illustra anche come le espressioni regolari possono usare stringhe codificate.

3.4 Espressioni regolari per la ricerca di configurazioni letterali

Molti compiti di elaborazione linguistica includono la ricerca di modelli lessicali. Per esempio, possiamo trovare parole che finiscono con *ed* usando `endswith('ed')`. Abbiamo visto una gamma di questi "word tests" nella Tabella 1.4. Le espressioni regolari ci forniscono un metodo più potente e flessibile per descrivere i modelli di carattere che ci interessano.

Nota

Sono state pubblicate molte altre introduzioni alle espressioni regolari, organizzate intorno alla sintassi delle espressioni regolari e applicate alla ricerca dei file di testo. Invece di farlo di nuovo, ci concentreremo sull'uso delle espressioni regolari a livelli differenti di elaborazione linguistica. Come sempre, adotteremo un approccio basato su un problema e presenteremo nuovi attributi solo se sono necessari per risolvere problemi pratici. Nel nostro discorso segneremo le espressioni regolari che usano le parentesi uncinate come questa: `<<patt>>`.

Per usare le espressioni regolari in Python dobbiamo importare la raccolta `re` usando: `import re`. Dobbiamo anche avere una lista di parole da cercare; useremo di nuovo il Word Corpus (Sezione 2.4). Dobbiamo preelaborarlo per rimuovere ogni nome proprio.

```
>>> import re
>>> wordlist = [w for w in nltk.corpus.words.words('en') if
w.islower()]
```

Usare Meta-Caratteri basici

Cerchiamo le parole che finiscono con *ed* usando l'espressione regolare «`ed$`». Useremo la funzione `re.search(p, s)` per verificare se il modello linguistico `p` può essere individuato da qualche parte della

stringa `s`. Dobbiamo specificare il carattere che ci interessa, e usare il simbolo del dollaro che ha un ruolo particolare nel contesto delle espressioni regolari poiché individua la fine della parola:

```
>>> [w for w in wordlist if re.search('ed$', w)]
['abaissed', 'abandoned', 'abased', 'abashed', 'abatised',
'abed', 'aborted', ...]
```

Il **metacarattere (wildcard) .** corrisponde ad ogni singolo carattere. Supponiamo di avere un cruciverba e di dover cercare una parola di otto lettere la cui terza lettera sia *j* e la sesta sia *t*. Al posto di ogni casella bianca usiamo un punto:

```
>>> [w for w in wordlist if re.search('^..j..t..$', w)]
['abjectly', 'adjuster', 'dejected', 'dejectly',
'injector', 'majestic', ...]
```

Nota

Tocca a te: Il simbolo di omissione ^ corrisponde all'inizio di una stringa, come il simbolo `$` corrisponde alla fine. Quali risultati avremo con l'esempio di cui sopra se omettiamo entrambi i simboli, e cerchiamo `<<..j..t..>>`?

Infine, il simbolo ? specifica che il carattere precedente è opzionale. Così «`^e-?mail$`» corrisponderà sia a *email* che a *mail*. Potremmo contare il numero totale delle occorrenze di questa parola (in entrambe le ortografie) in un testo usando `sum(1 for w in text if re.search('^e-?mail$', w))`.

Intervalli e chiusure

Figura 3.5: T9: Tex on 9 Keys

Il sistema **T9** è usato per comporre testi sui telefoni cellulari (vedi Figura 3.5). Due o più parole che vengono composte con la stessa sequenza di tasti sono detti **textonyms**. Per esempio, sia *hole* che *golf* sono inseriti premendo la sequenza 4653. Quali altre parole possono essere generate con la stessa sequenza? Qui usiamo l'espressione regolare «`^[ghi][mno][jlk][def]$`»:

```
>>> [w for w in wordlist if
re.search('^[ghi][mno][jlk][def]$', w)]
['gold', 'golf', 'hold', 'hole']
```

La prima parte dell'espressione, «`^[ghi]`», segna l'inizio di una parola seguita da *g*, *h*, o *i*. La parte successiva dell'espressione, «`[mno]`», vincola il secondo carattere ad essere *m*, *n* o *o*. Anche il terzo e il quarto carattere sono vincolati. Solo quattro parole soddisfano tutti questi vincoli. Notare che l'ordine dei caratteri nelle parentesi quadre non è rilevante, per cui potremmo aver scritto «`^[hig][nom][ljk][fed]$`» e combinato le stesse parole.

Nota

Tocca a te: Cercate alcune "finger-twisters", cercando parole che usano solo una parte della tastiera numerica. Per esempio `<<^[ghijklmno]+$>>`, o più sinteticamente, `<<^[g-o]+$>>`, troverà parole che usano solo i tasti 4, 5, 6 nella fila centrale, e `<<^ [a-fj-o]+$>>` troverà parole che usano 2, 3, 5, 6 nell'angolo in alto a destra. Cosa significano – e +?

Esploriamo il simbolo + un po' più a fondo. Notare che può essere applicato a lettere singole, o a serie di lettere tra parentesi:

```
>>> chat_words = sorted(set(w for w in
nltk.corpus.nps_chat.words()))
>>> [w for w in chat_words if re.search('^m+i+n+e+$', w)]
['miiiiiiiiiiiiinnnnnnnnnnneeeeeeeee',
'miiiiiinnnnnnnnneeeeeee', 'mine',
'mmmmmmmmiiiiiiiinnnnnnnneeeeeee']
>>> [w for w in chat_words if re.search('^[ha]+$', w)]
['a', 'aaaaaaaaaaaaaaa', 'aaahhhh', 'ah', 'ahah',
'ahahah', 'ahh',
'ahhahahaha', 'ahhh', 'ahhhh', 'ahhhhhh',
'ahhhhhhhhhhhhh', 'h', 'ha', 'haaa',
'hah', 'haha', 'hahaaa', 'hahah', 'hahaha', 'hahahaa',
'hahahah', 'hahahaha', ...]
```

Dovrebbe essere chiamo che + significa semplicemente "uno o più casi della stessa voce", che può essere un singolo carattere come `m`, una configurazione come `[fed]` o una gamma come `[d-f]`. Adesso sostituiamo + con * che significa "zero o più casi della stessa voce". L'espressione regolare `<<^m*i*n*e*$>>` troverà tutto quello che abbiamo trovato con `<<^m+i+n+e+$>>` , ma anche parole in cui alcune lettere non appaiono del tutto, es. *me*, *min*, e *mmmmm*. Notare che i simboli + e * sono di solito riferiti a **chiusure di Kleene**, o semplicemente **chiusure.**

L'operatore ^ ha un'altra funzione quando appare come primo carattere all'interno delle parentesi quadre. Per esempio `<<[^aeiouAEIOU]>>` trova ogni carattere diverso da una vocale. Possiamo cercare il NPS Chat Corpus per parole che sono costituite interamente da caratteri non vocalici usando `<<^[aeiouAEIOU]+$>>` per trovare elementi come questi: `:),:), :), grrr, cyb3r` e `zzzzzzzz`. Notare che ciò include anche caratteri non alfabetici.

Ecco alcuni altri esempi di espressioni regolari che vengono usati per cercare i token che corrispondono a un modello particolare, che illustrano l'uso di alcuni nuovi simboli: /, `{}`, `()`,e `|`:

```
>>> wsj = sorted(set(nltk.corpus.treebank.words()))
>>> [w for w in wsj if re.search('^[0-9]+\.[0-9]+$', w)]
```

```
['0.0085', '0.05', '0.1', '0.16', '0.2', '0.25', '0.28',
'0.3', '0.4', '0.5',
'0.50', '0.54', '0.56', '0.60', '0.7', '0.82', '0.84',
'0.9', '0.95', '0.99',
'1.01', '1.1', '1.125', '1.14', '1.1650', '1.17', '1.18',
'1.19', '1.2', ...]
>>> [w for w in wsj if re.search('^[A-Z]+\$$', w)]
['C$', 'US$']
>>> [w for w in wsj if re.search('^[0-9]{4}$', w)]
['1614', '1637', '1787', '1901', '1903', '1917', '1925',
'1929', '1933', ...]
>>> [w for w in wsj if re.search('^[0-9]+-[a-z]{3,5}$',
w)]
['10-day', '10-lap', '10-year', '100-share', '12-point',
'12-year', ...]
>>> [w for w in wsj if re.search('^[a-z]{5,}-[a-z]{2,3}-
[a-z]{,6}$', w)]
['black-and-white', 'bread-and-butter', 'father-in-law',
'machine-gun-toting',
'savings-and-loan']
>>> [w for w in wsj if re.search('(ed|ing)$', w)]
['62%-owned', 'Absorbed', 'According', 'Adopting',
'Advanced', 'Advancing', ...]
```

Nota

Tocca a te: Studiate gli esempi di sopra e provate ad capire cosa significano le notazioni /, {}, (), e | prima di continuare a leggere.

Probabilmente avrete capito che una barra retroversa significa che il carattere che segue è deprivato dei suoi poteri speciali e deve corrispondere letteralmente ad uno specifico carattere nella parola. Quindi, mentre . è speciale, `\.` corrisponde solo a un punto. Le espressioni nelle parentesi, come `{3,5}`, specificano il numero di ripetizioni dell'elemento precedente. La barra verticale (pipe character) indica una scelta tra il materiale alla sua sinistra o alla sua destra. Le parentesi indicano il campo di azione di un operatore: possono essere usate insieme alla barra verticale (o separatrice) come questo: `<<w(i|e|ai|oo)t>>` corrispondendo a *wit*, *wet*, *wait*, e *woot*. È istruttivo vedere cosa accade quando si omettono le parentesi dall'ultima espressione qui sopra, e cercare `<<ed|ing$>>`.

I meta-caratteri che abbiamo visto sono sintetizzati nella Tabella 3.3.

Tabella 3.3

Metacaratteri di base per le espressioni regolari, che includono wildcard, gamme e chiusure.

Operator	Behavior
`.`	Wildcard, matches any character
`^abc`	Matches some pattern *abc* at the start of a string
`abc$`	Matches some pattern *abc* at the end of a string
`[abc]`	Matches one of a set of characters
`[A-Z0-9]`	Matches one of a range of characters

`ed|ing|s`	Matches one of the specified strings (disjunction)
`*`	Zero or more of previous item, e.g. `a*`, `[a-z]*` (also known as *Kleene Closure*)
`+`	One or more of previous item, e.g. `a+`, `[a-z]+`
`?`	Zero or one of the previous item (i.e. optional), e.g. `a?`, `[a-z]?`
`{n}`	Exactly *n* repeats where n is a non-negative integer
`{n,}`	At least *n* repeats
`{,n}`	No more than *n* repeats
`{m,n}`	At least *m* and no more than *n* repeats
`a(b|c)+`	Parentheses that indicate the scope of the operators

Per l'interprete di Python, una espressione regolare è una stringa qualunque. Se la stringa contiene una barra retroversa seguita da particolari caratteri, l'interprete li tradurrà come speciali. Per esempio \b verrà interpretato come il carattere di ritorno indietro nel rigo. In generale, quando si usano espressioni regolari che contengono barre retroverse, dovremmo istruire l'interprete a non affatto guardare nella riga, ma semplicemente di passarla direttamente alla libreria `re` per l'elaborazione. Compiamo questa operazione prefissando la stringa con la lettera `r`, per indicare che questa è una **stringa aperta (raw string)**. Per esempio la stringa aperta `r'\band\b'` contiene due simboli `\b` che sono interpretati dalla libreria `re` come corrispondenti confini di parola invece che caratteri di rimando indietro. Se vi abituerete ad usare `r'...'` per le espressioni regolari- come faremo da qui in avanti- eviterete di pensare a queste complicazioni.

3.5 Applicazioni Utili delle Espressioni Regolari

Gli esempi sopra riportati implicavano tutti la ricerca di parole *w* che corrispondono una qualche espressione regolare tramite `re.search(regexp, w)`. A parte verificare se una espressione regolare corrisponde a una parola, possiamo usare le espressioni regolari per estrarre materiale dalle parole, o per modificare le parole in modi particolari.

Estrazione di pezzi di parole

Il metodo `re.findall()` ("cerca tutto") cerca tutte (non-sovrapposte) le corrispondenze dell'espressione regolare data. Cerchiamo tutte le vocali in una parole, poi contiamole:

```
>>> word = 'supercalifragilisticexpialidocious'
>>> re.findall(r'[aeiou]', word)
['u', 'e', 'a', 'i', 'a', 'i', 'i', 'i', 'e', 'i', 'a',
'i', 'o', 'i', 'o', 'u']
>>> len(re.findall(r'[aeiou]', word))
16
```

Cerchiamo tutte le sequenze di due o più vocali nello stesso testo, e determiniamo la loro relativa frequenza:

```
>>> wsj = sorted(set(nltk.corpus.treebank.words()))
```

```
>>> fd = nltk.FreqDist(vs for word in wsj
...                     for vs in
re.findall(r'[aeiou]{2,}', word))
>>> fd.items()
[('io', 549), ('ea', 476), ('ie', 331), ('ou', 329),
('ai', 261), ('ia', 253),
('ee', 217), ('oo', 174), ('ua', 109), ('au', 106),
('ue', 105), ('ui', 95),
('ei', 86), ('oi', 65), ('oa', 59), ('eo', 39), ('iou',
27), ('eu', 18), ...]
```

Nota

Tocca a te: Nel formato Data Time W3C, le date sono rappresentate così: 2009-12-31. Sostituiste il ? nel seguente codice Python con un'espressione regolare, al fine di convertire la stringa `'2009-12-31'` in una lista di numeri interi `[2009, 12, 31]`:

```
[int(n) for n in re.findall(?, '2009-12-31')]
```

Fare di più con pezzi di parole

Una volta che abbiamo usato `re.findall()` per estrarre materiale dalle parole, ci sono cose interessanti da fare con i pezzi, come incollarli di nuovo insieme o mischiarli.

È stato spesso notato che i testi in inglese sono spesso ridondanti, ed è tuttavia facile da leggere quando le vocali interne alle parole vengono lasciate fuori. Per esempio, *declaration* diventa *dclrtn*,e *inalienabile* diventa *inlnble*, conservando ogni vocale iniziale e finale. L'espressione regolare nel nostro prossimo esempio combina sequenze di vocali iniziali, sequenze di vocali finali, e tutte le consonanti; tutto il resto è ignorato. Questa disgiunzione a tre vie è elaborata sinistra-a-destra, se una delle tre parti combina una parola, le altre parti seguenti dell'espressione regolare sono ignorate. Usiamo `re.findall()` per estrarre tutti i pezzi combinati, e `''.join()` per unirli insieme (vedi la Sezione 3.9 per saperne di più sull'operazione di unione).

```
>>> regexp = r'^[AEIOUaeiou]+|[AEIOUaeiou]+$|[^AEIOUaeiou]'
>>> def compress(word):
...     pieces = re.findall(regexp, word)
...     return ''.join(pieces)
...
>>> english_udhr = nltk.corpus.udhr.words('English-Latin1')
>>> print nltk.tokenwrap(compress(w) for w in
english_udhr[:75])
Unvrsl Dclrtn of Hmn Rghts Prmble Whrs rcgntn of the inhrnt
dgnty and
of the eql and inlnble rghts of all mmbrs of the hmn fmly is
the fndtn
of frdm , jstce and pce in the wrld , Whrs dsrgrd and cntmpt
fr hmn
rghts hve rsltd in brbrs acts whch hve outrgd the cnscnce of
mnknd ,
and the advnt of a wrld in whch hmn bngs shll enjy frdm of
spch and
```

Successivamente, combiniamo le espressioni regolari con le distribuzioni di frequenza condizionale. Qui estraiamo tutte le sequenze consonante-vocale dalle parole della lingua Rotokas, come ad esempio *ka* e *si*.

Dal momento che ognuno di questi è un paio, può essere usato per inizializzare una distribuzione condizionata di frequenza. Poi tabuliamo la frequenza di ogni paio:

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
re.findall(r'[ptksvr][aeiou]', w)]
>>> cfd = nltk.ConditionalFreqDist(cvs)
>>> cfd.tabulate()
     a    e    i    o    u
k  418  148   94  420  173
p   83   31  105   34   51
r  187   63   84   89   79
s    0    0  100    2    1
t   47    8    0  148   37
v   93   27  105   48   49
```

Esaminando le righe per *s* e *t*, vediamo che sono in parziale "distribuzione complementare", il che è prova che non sono distinti fonemi nel linguaggio. Quindi, possiamo plausibilmente lasciar cadere la *s* dall'alfabeto Rotokas e semplicemente avere una regola di pronuncia che la lettera *t* venga pronunciata *s* quando seguita da una *i*. (Notare che il singolo lemma che ha *su*, cioè *kasuari*, 'cassowary' (casuario) è mutuato dall'inglese).

Se vogliamo essere capaci di verificare le parole dietro ai numeri nella tabella di sopra, potrebbe essere utile avere un indice, permettendoci di trovare velocemente la lista della parole che contiene una coppia consonante-vocale data, es. `cv_index['su']` dovrebbe darci tutte le parole che contengono *su*. Ecco come possiamo farlo:

```
>>> cv_word_pairs = [(cv, w) for w in rotokas_words
...                          for cv in
re.findall(r'[ptksvr][aeiou]', w)]
>>> cv_index = nltk.Index(cv_word_pairs)
>>> cv_index['su']
['kasuari']
>>> cv_index['po']
['kaapo', 'kaapopato', 'kaipori', 'kaiporipie', 'kaiporivira',
'kapo', 'kapoa',
'kapokao', 'kapokapo', 'kapokapo', 'kapokapoa', 'kapokapoa',
'kapokapora', ...]
```

Questo programma elabora ogni parola *w* alternativamente, e per ognuna, cerca ogni sottostringa che corrisponde all'espressione regolare «`[ptksvr][aeiou]`». Nel caso della parola *kasuari,* trova *ka, su* e ri. Perciò, la lista `cv_word_pairs` conterrà `('ka', 'kasuari'), ('su', 'kasuari') and ('ri', 'kasuari')`. Un ulteriore passo, usando `nltk.Index()`,lo converte in un indice utile.

Ricercare le radici della parola

Quando usiamo un motore di ricerca, di solito non ci importa (o al massimo notiamo) se le parole nel documento differiscono dai nostri termini di ricerca avendo finali differenti. Una richiesta per *laptops* trova documenti che contengono *laptop* e viceversa. Infatti, *laptop* e *laptops* sono solo due forme della stessa parola (o lemma). Per alcune attività di elaborazione di linguaggio vogliamo ignorare i finali di parola, e semplicemente continuare con radici di parola.

Ci sono molti modi in cui possiamo estrarre la radice di una parola. Ecco un approccio facile da ricordare che elimina qualsiasi cosa come i suffissi:

```
>>> def stem(word):
...     for suffix in ['ing', 'ly', 'ed', 'ious', 'ies',
'ive', 'es', 's', 'ment']:
...         if word.endswith(suffix):
...             return word[:-len(suffix)]
...     return word
```

Sebbene in definitiva useremo lo stemmer (stem, radice) proprio di NLTK, è interessante vedere come possiamo usare un'espressione regolare per questa applicazione. Il nostro primo passo è di costruire una separazione di tutti i suffissi. Dobbiamo racchiuderla in una parentesi al fine di limitare l'autonomia della disgiunzione.

```
>>> re.findall(r'^.*(ing|ly|ed|ious|ies|ive|es|s|ment)$',
'processing')
['ing']
```

Qui, `re.findall()` ci dà solo il suffisso sebbene l'espressione regolare combini l'intera parola. Questo accade perché le parentesi hanno una seconda funzione, di selezionare le sottostringhe che devono essere estratte. Se vogliamo usare le parentesi per specificare l'autonomia della disgiunzione, ma non per selezionare il materiale da produrre, dobbiamo aggiungere `?:`, che è solo una delle molte strane sottigliezze delle espressioni regolari. Ecco la versione rivista.

```
>>> re.findall(r'^.*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$',
'processing')
['processing']
```

Comunque, vorremmo in realtà dividere la parola in radice e suffisso. Perciò dovremmo sono parentesizzare entrambe le parti dell'espressione regolare:

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$',
'processing')
[('process', 'ing')]
```

Tutto ciò sembra promettente, ma ha ancora un problema. Guardiamo ad una parola diversa, *process:*

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$',
'processes')
[('processe', 's')]
```

L'espressione regolare ha trovato erroneamente un suffisso *–s* invece di un suffisso *–es*. Ciò dimostra un'altra sottigliezza: l'operatore asterisco è "avido" e la parte .* dell'espressione prova ad esaurire tanta parte dell'input quanto ne è possibile. Se usiamo la versione "non-avida" dell'operatore asterisco, scritta *?, otteniamo ciò che vogliamo:

```
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$',
'processes')
[('process', 'es')]
```

Questo funziona anche quanto accettiamo un suffisso vuoto, rendendo il contenuto seconde parentesi opzionale:

```
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$',
'language')
[('language', '')]
```

Questo approccio ha ancora molti problemi (puoi trovarli?) ma continueremo a definire una funzione che esegua lo stemming, e applicarla all'intero testo:

```
>>> def stem(word):
...     regexp = r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'
...     stem, suffix = re.findall(regexp, word)[0]
...     return stem
...
>>> 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 = nltk.word_tokenize(raw)
>>> [stem(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'women', 'ly', 'in',
'pond',
'distribut', 'sword', 'i', 'no', 'basi', 'for', 'a', 'system',
'of', 'govern',
'.', 'Supreme', 'execut', 'power', 'deriv', 'from', 'a',
'mandate', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcical',
'aquatic', 'ceremony', '.']
```

Notare che la nostra espressione regolare ha rimosso la *s* da *ponds* ma anche da *is* e *basis*. Produce alcune non-parole come *distribut* e *deriv*, ma queste sono radici accettabili in alcune applicazioni

Cercare testi tokenizzati

Potete usare un tipo speciale di espressione regolare per cercare attraverso parole multiple in un testo (dove il testo è una lista di token). Per esempio, `"<a> <man>"` trova tutti i casi di *a man* nel testo. Le parentesi uncinate sono usate per segnare i confini del token, e ogni spazio bianco tra le parentesi uncinate è ignorato (atteggiamento che è esclusiva del metodo per testi `findall()` di NLTK). Nell'esempio che segue, includiamo `<.*>` che combinerà ogni singolo token, e lo inseriremo nelle parentesi così che solo le parole combinate (es *monied*) e non le frasi combinate (es. *a monied man*) saranno prodotte. Il secondo esempio trova frasi da tre parole che finiscono con la parola *bro*. L'ultimo esempio trova sequenze di tre o più parole che cominciano con la lettera *l*.

```
>>> from nltk.corpus import gutenberg, nps_chat
>>> moby = nltk.Text(gutenberg.words('melville-
moby_dick.txt'))

>>> moby.findall(r"<a> (<.*>) <man>")
monied; nervous; dangerous; white; white; white; pious; queer;
good;
mature; white; Cape; great; wise; wise; butterless; white;
fiendish;
pale; furious; better; certain; complete; dismasted; younger;
```

```
brave;
brave; brave; brave
>>> chat = nltk.Text(nps_chat.words())

>>> chat.findall(r"<.*> <.*> <bro>")
you rule bro; telling you bro; u twizted bro

>>> chat.findall(r"<l.*>{3,}")
lol lol lol; lmao lol lol; lol lol lol; la la la la la; la la
la; la
la la; lovely lol lol love; lol lol lol.; la la la; la la la
```

Nota

Tocca a te: Consolidate il vostro apprendimento dei modelli di espressione regolari e delle sostituzioni usando `nltk.re_show (p, s)` che annota la stringa *s* a mostrare ogni punto in cui il modello *p* è combinato, e `nltk.app.nemo()` che fornisce una interfaccia grafica per l'esplorazione delle espressioni regolari. Per fare più pratica, provate alcuni esercizi sulle espressioni regolari alla fine di questo capitolo.

È facile costruire modelli di ricerca quando il fenomeno linguistico che stiamo studiando è legato a parole particolari. In alcuni casi, un po' di creatività garantirà il successo. Per esempio, cercare un vasto corpus di testi per le espressioni della forma *x and other ys* ci consente di scoprire gli iperonimi (Sezione 2.5):

```
>>> from nltk.corpus import brown
>>> hobbies_learned =
nltk.Text(brown.words(categories=['hobbies', 'learned']))
>>> hobbies_learned.findall(r"<\w*> <and> <other> <\w*s>")
speed and other activities; water and other liquids; tomb and
other
landmarks; Statues and other monuments; pearls and other
jewels;
charts and other items; roads and other features; figures and
other
objects; military and other areas; demands and other factors;
abstracts and other compilations; iron and other metals
```

Con abbastanza testo, questo approccio ci fornirà un utile raccolta di informazioni riguardo alla tassonomia degli oggetti, senza aver bisogno di nessun lavoro manuale. Comunque, i nostri risultati di ricerca conterranno di solito falsi positivi, es. casi che vorremmo escludere. Per esempio, il risultato *demands and other factors* suggerisce che *demand* è un'istanza del tipo *factor*, ma questa sentenza riguarda in realtà le domande salariali. Nonostante tutto, possiamo costruire la nostra ontologia dei concetti inglesi correggendo manualmente i risultati di queste ricerche.

Nota

Questa combinazione di elaborazioni automatiche e manuali è il modo più comune per costruire nuovi corpora. Ci ritorneremo nel Capitolo 11.

La ricerca dei corpora soffre anche del problema dei negativi falsi, es. omettendo i casi che vorremmo includere. È rischioso concludere che alcuni fenomeni linguistici non esistono in un corpus solo perché non abbiamo trovato istanze di un modello di ricerca. Forse non abbiamo pensato attentamente ai modelli adatti.

Nota

Tocca a te: Cercate esempi del modello *as x as y* per scoprire informazioni circa le entità e le loro proprietà.

3.6 Normalizzare il testo

Nei precedenti esempi di programma abbiamo spesso convertito il testo in minuscolo prima di fare altro con queste parole, es. `set(w.lower() for w in text)`. Usando `lower()`, abbiamo **normalizzato** il testo in minuscolo così che la distinzione tra *The* e *the* è ignorata. Spesso vogliamo fare altro rispetto a questo, e eliminare ogni affisso, una funzione conosciuta come stemming. Un passo ulteriore è essere sicuri che la forma che ne risulta è una parola nota in un dizionario, una funzione conosciuta come lemmatizzazione. Discuteremo ognuna di queste fasi di volta in volta. Per prima cosa, dobbiamo definire il dato 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 = nltk.word_tokenize(raw)
```

Stemmer

NLTK include alcuni stemmer prodotti in serie, e semmai doveste aver bisogno di uno stemmer dovrete usare uno di questi a piacere per produrne a mano uno proprio usando le espressioni regolari, dal momento che azionano una vasta gamma di casi irregolari. Gli stemmer Porter e Lancaster seguono le loro regole per l'asportazione degli affissi. Osservate che lo stemmer Porter azione correttamente la parola *liyng* (mappandolo verso *lie*), mentre lo stemmer Lancaster non lo fa.

```
>>> 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', '.']
```

Lo stemming non è un processo ben definito, e generalmente scegliamo lo stemmer che meglio si applica alla operazione che abbiamo in mente. Il Porter Stemmer è una buona scelta se si stanno indicizzando alcuni testi e si vuole supportare la ricerca usando forme e parole alternative (illustrate nell'Esempio 3.6, che usa

tecniche di programmazione orientate agli oggetti che sono al di là dello scopo di questo libro, tecniche di formattazione delle stringhe che saranno trattati nella Sezione 3.9, e la funzione `enumerate()` che verrà spiegata nella Sezione 4.2).

```
class IndexedText(object):

    def __init__(self, stemmer, text):
        self._text = text
        self._stemmer = stemmer
        self._index = nltk.Index((self._stem(word), i)
                                 for (i, word) in
enumerate(text))

    def concordance(self, word, width=40):
        key = self._stem(word)
        wc = width/4                # words of context
        for i in self._index[key]:
            lcontext = ' '.join(self._text[i-wc:i])
            rcontext = ' '.join(self._text[i:i+wc])
            ldisplay = '%*s'  % (width, lcontext[-width:])
            rdisplay = '%-*s' % (width, rcontext[:width])
            print ldisplay, rdisplay

    def _stem(self, word):
        return self._stemmer.stem(word).lower()
```

```
>>> porter = nltk.PorterStemmer()
>>> grail = nltk.corpus.webtext.words('grail.txt')
>>> text = IndexedText(porter, grail)
>>> text.concordance('lie')
r king ! DENNIS : Listen , strange women lying in ponds
distributing swords is no
 beat a very brave retreat . ROBIN : All lies ! MINSTREL
: [ singing ] Bravest of
       Nay . Nay . Come . Come . You may lie here . Oh ,
but you are wounded !
doctors immediately ! No , no , please ! Lie down . [
clap clap ] PIGLET : Well
ere is much danger , for beyond the cave lies the Gorge
of Eternal Peril , which
   you . Oh ... TIM : To the north there lies a cave --
the cave of Caerbannog --
h it and lived ! Bones of full fifty men lie strewn about
its lair . So , brave k
not stop our fight ' til each one of you lies dead , and
the Holy Grail returns t
```

Esempio 3.6 (code_stemmer_indexing.py): Indicizzare un testo usando uno Stemmer

Lemmatizzazione

Il lemmatizzatore WordNet rimuove solo gli affissi se la parola che risulta è nel suo dizionario. Questo processo addizionale di controllo rende il lemmatizzatore più lento dei precedenti stemmer. Notare che non gestisce *lying*, ma converte *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 lemmatizzatore WordNet è un'ottima scelta se si vuol stilare il vocabolario di alcuni testi e se si vuole una lista di lemmi validi (o di lemmi lessicali).

Nota

Un'altra funzione di normalizzazione riguarda l'identificazione di **parole non-standard** inclusi i numeri, le abbreviazioni, e le date, e mappa ogni sorta di token in una specie di dizionario. Per esempio, ogni numero decimale può essere mappato in un singolo token `0.0` , e ogni acronimo può essere mappato in AAA. Ciò rende il vocabolario più piccolo e aumento l'accuratezza di molte funzioni di modellamento del linguaggio.

3.7 Espressioni regolari per la tokenizzazione del testo

La tokenizzazione è la funzione di tagliare una stringa in unità linguistiche identificabili che costituiscono un pezzo di dati linguistici. Sebbene sia una funzione fondamentale, siamo stati in grado di eliminarla fino ad adesso perché molti corpora sono già tokenizzati, e perché NLTK include alcuni tokenizzatori. Adesso che le espressioni regolari ci sono familiari, possiamo imparare come usarle per tokenizzare il testo, e per avere ancora più controllo sul processo.

Approcci semplici alla tokenizzazione

Il metodo più semplice per tokenizzare un testo è dividere per spazi. Considerate il seguente testo preso da *Alice's Adventures in Wonderland:*

```
>>> raw = """'When I'M a Duchess,' she said to herself,
(not in a very hopeful tone
... though), 'I won't have any pepper in my kitchen AT
ALL. Soup does very
... well without--Maybe it's always pepper that makes
people hot-tempered,'..."""
```

Possiamo divider questo testo semplice in spazi usando la funzione `raw.split()`. Per fare la stessa cosa usando un'espressione regolare, non è sufficiente sfruttare ogni carattere di spaziatura nella stringa dal momento che questa risulta in tokens che contiene il carattere `\n` di nuova riga; al contrario dobbiamo contrassegnare ogni numero di spazi, tabulatori, o nuove righe.

```
>>> re.split(r' ', raw)
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to',
'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone\nthough),', "'I", "won't",
'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does',
'very\nwell', 'without--Maybe',
"it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-
tempered,'..."]

>>> re.split(r'[ \t\n]+', raw)
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to',
'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone', 'though),', "'I", "won't",
'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very',
'well', 'without--Maybe',
"it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-
tempered,'..."]
```

L'espressione regolare «`[ \t\n]+`» segna uno o più spazi, tabulatori (`\t`) o nuova riga (`\n`). Gli altri caratteri di spaziatura, come ritorno a capo e form-feed (terminare la pagina sulla stampante e avanzare al modulo successivo, n.d.t) dovrebbero essere effettivamente inclusi allo stesso modo. Invece noi useremo un'abbreviazione già inserita, `\s`, che indica ogni carattere di spaziatura. L'istruzione sopra riportata può essere riscritta come `re.split(r'\s+', raw)`.

Nota
Importante: ricordate di prefissare le espressioni regolari con la lettera `r` (che significa "raw"), che suggerisce all'interprete Python di trattare la stringa letteralmente, piuttosto che elaborare ogni barra retroversa che contiene.

Dividere per spazi ci dà token come `'(not'` e `'herself,'`. Un'alternativa è usare il fatto che Python ci fornisce una classe di carattere `\w` per caratteri parola, equivalenti di `[a-zA-ZO-9_]`. Definisce anche il complemento di questa classe `\W`, es. tutti i caratteri diversi dalle lettere, numeri e underscore. Possiamo usare \W in una semplice espressione regolare per dividere l'input in qualcosa di diverso da un carattere parola:

```
>>> re.split(r'\W+', raw)
['', 'When', 'I', 'M', 'a', 'Duchess', 'she', 'said',
'to', 'herself', 'not', 'in',
'a', 'very', 'hopeful', 'tone', 'though', 'I', 'won',
't', 'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL', 'Soup', 'does',
'very', 'well', 'without',
'Maybe', 'it', 's', 'always', 'pepper', 'that', 'makes',
'people', 'hot', 'tempered',
'']
```

Osservate che ciò ci dà una stringa vuota all'inizio e alla fine (per capire perché, provate ad applicare `'xx'.split('x')`). Abbiamo gli stessi token, ma senza stringhe vuote, con `re.findall(r'\w+', raw)`, usando un modello che contrassegna le parole invece degli spazi. Adesso che stiamo contrassegnando

le parole, siamo nella posizione di estendere l'espressione regolare per coprire una più vasta gamma di casi. L'espressione regolare «`\w+|\S\w*`» per prima cosa proverà a contrassegnare ogni sequenza di caratteri di parola. Se non è stata trovata nessuna corrispondenza, proverà a contrassegnare ogni carattere di non spaziatura (\S è il complemento di \s) seguito da ulteriori caratteri di parola. Ciò significa che la punteggiatura è raggruppata con tutte le altre lettere che seguono (es. 's) ma quelle sequenze di due o più segni di punteggiatura sono divise.

```
>>> re.findall(r'\w+|\S\w*', raw)
["'When", 'I', "'M", 'a', 'Duchess', ',', "'", 'she', 'said',
'to', 'herself', ',',
'(not', 'in', 'a', 'very', 'hopeful', 'tone', 'though', ')',
',', "'I", 'won', "'t",
'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL',
'.', 'Soup', 'does',
'very', 'well', 'without', '-', '-Maybe', 'it', "'s",
'always', 'pepper', 'that',
'makes', 'people', 'hot', '-tempered', ',', "'", '.', '.',
'.']
```

Generalizziamo il `\w+` nella espressione di sopra per autorizzare i trattini all'interno della parola e gli apostrofi : «`\w+([-']\w+)*`». Questa espressione significa \w+ seguita da zero o più istanze di `[-']\w+`; corrisponderà a *hot-tempered* e *it's*. (Dobbiamo includere `?:` in questa espressione per motivi discussi precedentemente.) Aggiungeremo anche un modello per contrassegnare le virgolette così che queste saranno tenute separate dal testo che racchiudono.

```
>>> print re.findall(r"\w+(?:[-']\w+)*|'|[-.(]+|\S\w*",
raw)
["'", 'When', "I'M", 'a', 'Duchess', ',', "'", 'she',
'said', 'to', 'herself', ',',
'(', 'not', 'in', 'a', 'very', 'hopeful', 'tone',
'though', ')', ',', "'", 'I',
"won't", 'have', 'any', 'pepper', 'in', 'my', 'kitchen',
'AT', 'ALL', '.', 'Soup',
'does', 'very', 'well', 'without', '--', 'Maybe', "it's",
'always', 'pepper',
'that', 'makes', 'people', 'hot-tempered', ',', "'",
'...']
```

L'espressione di cui sopra include anche «`[-.(]+`» che genera anche il doppio trattino, ellissi, e parentesi aperte da tokenizzare separatamente.

Tabella 3.4 lista della classe di simboli delle espressioni regolari che abbiamo visto in questa sezione, con l'aggiunta di altri simboli utili.

Tabella 3.4:

Simboli di espressione regolare

Symbol	Function
`\b`	Word boundary (zero width)
`\d`	Any decimal digit (equivalent to `[0-9]`)

`\D`	Any non-digit character (equivalent to `[^0-9]`)
`\s`	Any whitespace character (equivalent to `[ \t\n\r\f\v]`
`\S`	Any non-whitespace character (equivalent to `[^ \t\n\r\f\v]`)
`\w`	Any alphanumeric character (equivalent to `[a-zA-Z0-9_]`)
`\W`	Any non-alphanumeric character (equivalent to `[^a-zA-Z0-9_]`)
`\t`	The tab character
`\n`	The newline character

Tokenizzatore NLTK delle espressioni regolari

La funzione `nltk.regexp_tokenize()` è simile a `re.findall()` (come lo abbiamo usato per la tokenizzazione). Comunque, `nltk.regexp_tokenize()` è più efficiente per questo compito, ed evita il bisogno delle parentesi per mansioni particolari. Per maggiore leggibilità divideremo l'espressione regolare in alcune righe e aggiungeremo un commento ad ogni riga. Lo speciale "contrassegno dettagliato" (verbose flag) `(?x)` dice a Python di togliere lo spazio bianco e i commenti incorporati.

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal
hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g.
$12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Quando usiamo i contrassegni dettagliati, non si possono più usare `' '` per contrassegnare un carattere di spaziatura; invece si può usare `\s`. La funzione `regexp_tokenize()` ha un parametro `gaps` opzionale. Quando assegnato a `True`, l'espressione regolare specifica la differenza tra token, come con `re.split()`.

Nota
Possiamo valutare un tokenizzatore comparando i token che risultato con una lista di parole, e ripostando ogni token che non appare nella lista di parole, usando `set(tokens).difference(wordlist)`. Probabilmente per prima cosa vorreste mettere in minuscolo tutti i token.

Ulteriori esiti della Tokenizzazione

La tokenizzazione si rivela essere un po' più complicata di quanto ci si aspettava. Nessuna singola soluzione funziona bene a tutti i livelli, e dobbiamo decidere cosa conta quanto un token in base al dominio di applicazione .

Quando si sviluppa un tokenizzatore può essere di aiuto accedere a testi semplici che sono stati tokenizzati manualmente, al fine di comparare il risultato del vostro tokenizzatore con token di alta qualità (o "standard-d'oro"). Il corpus di NLTK include un campione di dati del Penn Treebank, incluso il testo semplice del Wall Street Journal (`nltk.corpus.treebank_raw.raw()`) e la versione tokenizzata (`nltk.corpus.treebank.words()`).

Un ultimo problema per la tokenizzazione è la presenza di contrazioni, come ad esempio *didn't.* se stiamo analizzando il significato di una frase, sarà sicuramente più utile normalizzare questa forma in due forme separate: *did* e *n't* (o *not*). Possiamo compiere questo lavoro con l'aiuto di una tabella di ricerca.

3.8 Segmentazione

Questa sezione tratta concetti più avanzati, che potete decidere di saltare per la prima volta attraverso questo capitolo.

La tokenizzazione è un'istanza di problemi più generali di **segmentazione**. In questa sezione guarderemo due altre istanze di questo problema, che usano tecniche radicalmente diverse da quella che abbiamo visto finora in questo capitolo.

Segmentazione delle sentenze

La manipolazione di un testo al livello delle singole parole presuppone l'abilità di dividere il testo in singole frasi. Come abbiamo già visto, alcuni corpora forniscono già l'accesso al livello delle frasi. Nell'esempio che segue, calcoleremo il numero medio di parole per frase nel Brown Corpus:

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

In altri casi, il testo è disponibile solo come un flusso di caratteri. Prima di tokenizzare il testo in parole, abbiamo bisogno di segmentarlo in frasi. NLTK facilita questo compito includendo il segmentatore di frasi Punkt (Kiss & Strunk. 2006). Qui un esempio dei suoi usi nella segmentazione del testo di un romanzo. (Notare che se i dati interni del segmentatore sono stati aggiornati dal momento in cui leggete questo, avrete un risultato diverso):

```
>>>
sent_tokenizer=nltk.data.load('tokenizers/punkt/english.pickle')
>>> text = nltk.corpus.gutenberg.raw('chesterton-thursday.txt')
>>> sents = sent_tokenizer.tokenize(text)
>>> pprint.pprint(sents[171:181])
['"Nonsense!',
 '" said Gregory, who was very rational when anyone
else\nattempted paradox.',
 '"Why do all the clerks and navvies in the\nrailway trains look
so sad and tired,...',
```

```
 'I will\ntell you.',
 'It is because they know that the train is going right.',
 'It\nis because they know that whatever place they have taken a
ticket\nfor that ...',
 'It is because after they have\npassed Sloane Square they know
that the next stat...',
 'Oh, their wild rapture!',
 'oh,\ntheir eyes like stars and their souls again in Eden, if
the next\nstation w...'
 '"\n\n"It is you who are unpoetical," replied the poet Syme.']
```

Notare che questo esempio è effettivamente una singola frase, che riporta il discorso di Mr Lucian Gregory. Comunque, il discorso virgolettato contiene alcune frasi, e queste sono state divise in singole stringhe. Questo è il modo più logico per la maggior parte delle applicazioni.

La segmentazione della frase è complessa perché il punto è usato per segnare le abbreviazioni, e alcuni punti segnano simultaneamente una abbreviazione e terminano una frase, come spesso accade con gli acronimi come *U.S.A.*

Per un altro approccio alla segmentazione della frase, vedere la Sezione 6.2.

Segmentazione della parola

Per alcuni sistemi di scrittura, tokenizzare il testo è reso ancora più difficile dal fatto che non ci sono rappresentazioni visuali dei confini di parola. Per esempio, in Cinese, la stringa da tre caratteri: 爱国人 (ai4 "love" (verb), guo3 "country", ren2 "person") può essere tokenizzata come爱国 / 人, "country-loving person" or as 爱 / 国人, "love country-person."

Un problema del genere si solleva nell'elaborazione del linguaggio parlato, in cui l'ascoltatore deve segmentare un flusso di discorso ininterrotto in singole parole. Una versione particolarmente avvincente di questo problema si solleva quando non conosciamo le parole in anticipo. Questo è il problema che affronta chi impara la lingua, come un bambino che ascolta le parole pronunciate dai genitori. Considerare il seguente esempio artificiale, in cui i confini delle parole sono stati rimossi:

(1)
a. doyouseethekitty
b. seethedoggy
c. doyoulikethekitty
d. likethedoggy

La nostra prima impresa è semplicemente quella di rappresentare il problema: abbiamo bisogno di trovare un modo per separare il contenuto del testo dalla segmentazione. Possiamo fare ciò segnandoci ogni carattere con un valore booleano (boolean value) per indicare se un separatore di parole appare dopo il carattere (un'idea che può essere usata abbondantemente per il "chuncking" nel Capitolo 7). Ammettiamo che l'allievo tiene conto delle interruzioni di enunciazione, dal momento che queste corrispondono a pause estese. C'è una possibile rappresentazione, incluse le segmentazioni iniziali e di obiettivo:

```
>>> text =
"doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 =
"000000000000001000000000100000000000000010000000000"
>>> seg2 =
```

```
"010010010010000100100100001010010001001000010001001000010"
```

Osservate che le stringhe di segmentazione sono composte da zero e uno. Si tratta di un carattere in meno rispetto al testo di partenza, dal momento che un testo di lunghezza *n* può essere suddiviso in *n-l* punti. La funzione nell'Esempio 3.7 dimostra che possiamo tornare all'originale testo segmentato dalla rappresentazione di sopra.

```
def segment(text, segs):
    words = []
    last = 0
    for i in range(len(segs)):
        if segs[i] == '1':
            words.append(text[last:i+1])
            last = i+1
    words.append(text[last:])
    return words
```

```
>>> text = "doyouseethekittyseethedoggydoyouliketheKittyliketheDoggy"
>>> seg1 = "000000000000000100000000001000000000000000000100000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001001000010"
>>> segment(text, seg1)
['doyouseethekitty', 'seethedoggy', 'doyouliketheKitty',
'liketheDoggy']
>>> segment(text, seg2)
['do', 'you', 'see', 'the', 'kitty', 'see', 'the', 'doggy', 'do',
'you',
 'like', 'the', kitty', 'like', 'the', 'doggy']
```

Esempio 3.7 (code_segment.py): **Figura 3.7**:Ricostruire il testo segmentato dalla rappresentazione della stringa: `seg1` e `seg2` rappresentano le segmentazioni iniziali e finali di alcuni ipotetici discorsi diretti di un bambino; la funzione `segment()` può usarli per riprodurre il testo segmentato.

Adesso la funzione di segmentazione diventa un problema di ricerca: trovate la stringa bit che faccia in modo che il testo della stringa sia correttamente segmentato in parole. Supponiamo che l'allievo stia imparando le parole e le costituisca in un lessico interno. Fornendo un lessico applicabile, è possibile ricostruire il testo sorgente come una sequenza di elementi lessicali. Continuando (Brent, 1995), possiamo definire una **funzione obiettivo**, una funzione score di cui proveremo ad ottimizzare i valori, basata sulla grandezza del lessico e sulla quantità di informazioni che servono per ricostruire il testo sorgente dal lessico. Lo illustriamo nella Figura 3.8.

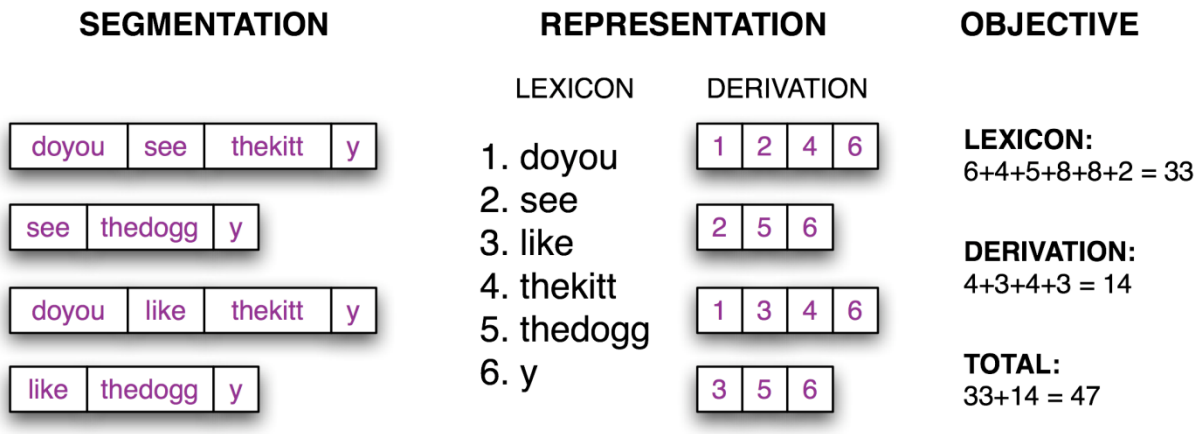


Figura 3.8: Calcolo della funzione obiettivo: data un'ipotetica segmentazione del testo sorgente (sulla sinistra), derivare una tabella del lessico e di derivazione che permetta di ricostruire il testo sorgente, poi sommare il numero dei caratteri usati da ogni elemento lessicale (incluso il marcatore di confini) e ogni derivazione, per funzionare come risultato della qualità della segmentazione; un valore del risultato più piccolo indica una migliore segmentazione.

È un lavoro semplice implementare questa funzione obiettivo, come mostrato nell'Esempio 3.9.

```
def evaluate(text, segs):
    words = segment(text, segs)
    text_size = len(words)
    lexicon_size = len(' '.join(list(set(words))))
    return text_size + lexicon_size
```

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "000000000000000100000000001000000000000000100000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001010000"
>>> seg3 = "000010010000001100100000011000010001000000110001000001"
>>> segment(text, seg3)
['doyou', 'see', 'thekitt', 'y', 'see', 'thedogg', 'y', 'doyou',
'like',
 'thekitt', 'y', 'like', 'thedogg', 'y']
>>> evaluate(text, seg3)
46
>>> evaluate(text, seg2)
47
>>> evaluate(text, seg1)
63
```

Esempio 3.9 (code_evaluate.py): Figura 3.9: Calcolare il costo di immagazzinamento del lessico e di ricostruzione del testo sorgente.

Il passo finale è di cercare degli schemi di zeri e uno che minimizzano questa funzione obiettivo, mostrata nell'Esempio 3.10. notare che la migliore segmentazione include "parole" come *thekitty*, dal momento che non c'è abbastanza prova nel dato per dividerla ulteriormente.

```
from random import randint

def flip(segs, pos):
    return segs[:pos] + str(1-int(segs[pos])) + segs[pos+1:]

def flip_n(segs, n):
    for i in range(n):
        segs = flip(segs, randint(0,len(segs)-1))
    return segs

def anneal(text, segs, iterations, cooling_rate):
    temperature = float(len(segs))
    while temperature > 0.5:
        best_segs, best = segs, evaluate(text, segs)
        for i in range(iterations):
            guess = flip_n(segs, int(round(temperature)))
            score = evaluate(text, guess)
            if score < best:
                best, best_segs = score, guess
        score, segs = best, best_segs
        temperature = temperature / cooling_rate
        print evaluate(text, segs), segment(text, segs)
    print
    return segs
```

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "0000000000000001000000000010000000000000000100000000000"
>>> anneal(text, seg1, 5000, 1.2)
60 ['doyouseetheki', 'tty', 'see', 'thedoggy', 'doyouliketh',
'ekittylike', 'thedoggy']
58 ['doy', 'ouseetheki', 'ttysee', 'thedoggy', 'doy', 'o',
'ulikethekittylike', 'thedoggy']
56 ['doyou', 'seetheki', 'ttysee', 'thedoggy', 'doyou', 'liketh',
'ekittylike', 'thedoggy']
54 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou',
'likethekittylike', 'thedoggy']
53 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'like',
'thekitty', 'like', 'thedoggy']
51 ['doyou', 'seethekittysee', 'thedoggy', 'doyou', 'like', 'thekitty',
'like', 'thedoggy']
42 ['doyou', 'see', 'thekitty', 'see', 'thedoggy', 'doyou', 'like',
'thekitty', 'like', 'thedoggy']
'000010010000000100100000001000010001000000010001000000'
```

Esempio 3.10 (code_anneal.py): Figura 3.10: Ricerca non deterministica che usa una ricottura simulata (Simulated Annealing): cominciare cercando solo con segmentazioni di frase; in modo casuale sconvolgete il proporzionale di zeri e uno alla "temperatura"; con ogni iterazione la temperatura è abbassata e la perturbazione del confini è ridotta.

Con abbastanza dati, è possibile segmentare automaticamente il testo in parole con un ragionevole grado di accuratezza. Questi metodi possono essere applicati alla tokenizzazione per sistemi di scrittura che non hanno nessuna rappresentazione visuale dei confini di parola.

3.9 Formattare: dalle liste alle stringhe

Spesso scriviamo un programma per riportare un singolo elemento, come ad esempio un elemento in particolare in un corpus che incontra alcuni criteri complicati, o un singolo riassunto statistico come un conteggio delle parole o la prestazione di un tagger. Più spesso, scriviamo un programma per produrre un risultato strutturato; per esempio, una tabulazione di numeri o forme linguistiche, o una riformattazione dei dati originali. Quando i risultati da presentare sono linguistici, il rendimento testuale è la scelta più naturale. Comunque, quando i risultati sono numerici, può essere preferibile produrre output grafici. In questa sezione apprenderete della varità di modi di presentare un output di programma.

Dalle liste alle stringhe

Il tipo più semplice di oggetto strutturato che usiamo per l'elaborazione dei testi è la lista di parole. Quando vogliamo restituirli su un display o in un file, dobbiamo convertire queste liste in stringhe. Per fare ciò con Python usiamo il metodo `join()` , e specifichiamo la stringa da usare come "colla".

```
>>> silly = ['We', 'called', 'him', 'Tortoise', 'because', 'he', 'taught',
'us', '.']
>>> ' '.join(silly)
'We called him Tortoise because he taught us .'
>>> ';'.join(silly)
'We;called;him;Tortoise;because;he;taught;us;.'
>>> ''.join(silly)
'WecalledhimTortoisebecausehetaughtus.'
```

In questo modo `' '.join(silly)` significa: prendere tutti gli elementi in `silly` e concatenarli come un'unica stringa, usando `' '` come spaziatore tra gli elementi. Il metodo `join()` lavora solo su una lista di stringhe- quello che abbiamo chiamato un testo- un tipo complesso che gode di alcuni privilegi in Python.

Stringhe e formati

Abbiamo visto che ci sono due modi di visualizzare il contenuto di un oggetto:

```
>>> word = 'cat'
>>> sentence = """hello
... world"""
>>> print word
cat
>>> print sentence
hello
world
>>> word
'cat'
>>> sentence
'hello\nworld'
```

Il commando `print` concede a Python la possibilità di produrre la forma più umanamente leggibile di un oggetto. Il secondo metodo- chiamare la variabile nel prompt dei comandi- ci mostra una stringa che può essere usata per ricreare questo oggetto. È importante tenere a mente che entrambe sono solo stringhe, visualizzate a vantaggio dell'utente. Non ci danno alcun indizio sull'effettiva rappresentazione interna dell'oggetto.

Ci sono molti altri utili modi per visualizzare un oggetto come stringa di caratteri. Ciò può avvenire per il beneficio del lettore umano, o perché vogliamo **esportare** i nostri dati su particolare formato di file per usarlo in un programma esterno. Il rendimento formattato tipicamente contiene una combinazioni di variabili e stringhe pre-specificate, es. data una distribuzione di frequenza `fdist` possiamo:

```
>>> fdist = nltk.FreqDist(['dog', 'cat', 'dog', 'cat', 'dog', 'snake',
'dog', 'cat'])
>>> for word in fdist:
...     print word, '->', fdist[word], ';',
dog -> 4 ; cat -> 3 ; snake -> 1 ;
```

al di là del problema degli spazi indesiderati, stampare dichiarazioni che contengono variabili e costanti alternate può essere difficile da leggere e mantenere. Una soluzione migliore è usare le **espressioni di formattazione di stringa.**

```
>>> for word in fdist:
...     print '%s->%d;' % (word, fdist[word]),
dog->4; cat->3; snake->1;
```

Per capire cosa sta accadendo, testiamo l'espressione di formattazione della stringa al suo interno. (Da adesso questo sarà il vostro nuovo metodo per esplorare la sintassi).

```
>>> '%s->%d;' % ('cat', 3)
'cat->3;'
>>> '%s->%d;' % 'cat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

I simboli speciali `%s` e `%d` sono variabili metasintattiche per stringhe e numeri interi (decimali). Possiamo racchiuderli in una stringa, poi usare l'operatore `%` per combinarli. Facciamo comprimere ulteriormente il codice di cui sopra, al fine di vedere di vedere questo comportamento da vicino:

```
>>> '%s->' % 'cat'
'cat->'
>>> '%d' % 3
'3'
>>> 'I want a %s right now' % 'coffee'
'I want a coffee right now'
```

Possiamo avere un numero di variabili metasintattiche, ma seguendo l'operatore `%` abbiamo bisogno di specificare un tupla con esattamente lo stesso numero di valori.

```
>>> "%s wants a %s %s" % ("Lee", "sandwich", "for lunch")
'Lee wants a sandwich for lunch'
```

Possiamo anche provvedere a valori per le variabili meta sintattiche indirettamente.. ecco un esempio che usa una congiunzione `for`:

```
>>> template = 'Lee wants a %s right now'
>>> menu = ['sandwich', 'spam fritter', 'pancake']
>>> for snack in menu:
...     print template % snack
...
Lee wants a sandwich right now
Lee wants a spam fritter right now
Lee wants a pancake right now
```

I simboli %s e %d sono chiamati **specificatori di conversione**. Cominciano con il carattere `%` e finiscono con un carattere di conversione come `s` (per la stringa) o `d` (per i numeri interi decimali). La stringa che contiene lo specificatore di conversione è chiamata **stringa di formato**. Combiniamo una stringa di formato con l'operatore e una tupla di valori per creare una espressione completa di stringa di formattazione.

Allineare le cose

Finora le nostre stringhe di formattazione hanno generato rimandi di ampiezza arbitraria sulla pagina (o sullo schermo), come `%s` e `%d`. Possiamo specificare l'ampiezza come più ci pare, come ad esempio `%6s`, producendo una stringa che è riempita di ampiezza 6. È giustificato a destra dal default, ma possiamo includere il segno meno per far sì che diventi giustificato sulla sinistra. Nel caso non sapessimo in anticipo quanto dovrebbe essere largo il valore visualizzato, il valore d'ampiezza può essere sostituito con un asterisco nella stringa di formattazione, per poi specificare usando una variabile.

```
>>> '%6s' % 'dog'
'   dog'

>>> '%-6s' % 'dog'
'dog   '
>>> width = 6

>>> '%-*s' % (width, 'dog')
'dog   '
```

Gli altri caratteri di controllo sono usati per i numeri interi decimali e per i numeri in virgola mobile. Dal momento che il carattere di percentuale `%` ha un'interpretazione speciale nelle stringhe di formattazione, dobbiamo farlo precedere da un altro `%` per ottenerlo come risultato.

```
>>> count, total = 3205, 9375
>>> "accuracy for %d words: %2.4f%%" % (total, 100 * count / total)
'accuracy for 9375 words: 34.1867%'
```

Un uso importante delle stringhe di formattazione è per la tabulazione dei dati. Ricordiamo che nella Sezione 2.1 abbiamo visto i dati che venivano tabulati da una frequenza di distribuzione condizionale. Operiamo la tabulazione da noi, facendo molta attenzione alle intestazioni e alla larghezza delle colonne, come mostrato nell'Esempio 3.11. notare la netta separazione tra il lavoro di elaborazione del linguaggio, e la tabulazione dei risultati.

```
def tabulate(cfdist, words, categories):
    print '%-16s' % 'Category',
    for word in words:                                  # column
headings
        print '%6s' % word,
    print
    for category in categories:
        print '%-16s' % category,                       # row heading
        for word in words:                              # for each word
            print '%6d' % cfdist[category][word],       # print table
cell
        print                                           # end the row
```

```
>>> from nltk.corpus import brown
>>> 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']
>>> tabulate(cfd, modals, genres)
Category            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     43
humor                16     30      8      8      9     13
```

Esempio 3.11 (code_modal_tabulate.py): **Figura 3.11**: Frequenza dei Modali nelle diverse sezioni del Brown Corpus

Ricordiamo dalla lista nell'Esempio 3.6 che abbiamo usato una stringa di formattazione `"%*s"`. Questo ci permette di specificare l'ampiezza di un campo usando una variabile.

```
>>> '%*s' % (15, "Monty Python")
'   Monty Python'
```

Possiamo fare ciò per personalizzare automaticamente le colonne in modo che siano grandi abbastanza da sistemare tutte le parole, usando `width = max(len(w) for w in words)`. Ricordare che la virgola alla fine del comando print aggiunge uno spazio ulteriore, e ciò è sufficiente a prevenire che le intestazioni della colonna vadano da una parte all'altra.

Scrivere il risultato in un File

Abbiamo visto come leggere un testo da un file (Sezione 3.1). Spesso è utile scrivere l'output sul file. Il seguente codice apre un file in scrittura, e salva il programma di uscita sul file.

```
>>> output_file = open('output.txt', 'w')
>>> words = set(nltk.corpus.genesis.words('english-kjv.txt'))
>>> for word in sorted(words):
...     output_file.write(word + "\n")
```

Nota
Tocca a te: Qual è l'effetto di aggiungere `\n` a ogni stringa prima di scriverla sul file? Se stiamo usando un sistema operativo Windows vorreste invece usare `word+"\r\n"`. Cosa succede se inseriamo `output_file.write(word)` ?

Quando scriviamo dati non di testo in un file dobbiamo convertirli per prima cosa in una stringa. Operiamo questa conversione usando le stringhe di formattazione, come abbiamo visto sopra. Scriviamo il numero totale delle parole nel nostro file, prima di chiuderlo.

```
>>> len(words)
2789
>>> str(len(words))
'2789'
>>> output_file.write(str(len(words)) + "\n")
>>> output_file.close()
```

Attenzione!

Dovreste evitare nomi di file che contengono caratteri di spaziatura come `output file.txt`, o che sono identici eccetto per i casi distinti, es. `Output.txt` e `output.TXT`.

Disposizione del testo

Quando l'output del nostro programma è in formato letterale, invece che tabulare, solitamente sarà necessario disporlo in modo che possa essere visualizzato in modo agevole. Considerare il seguente output, che sconfina dalla sua riga, e che usa un complicato comando `print` :

```
>>> saying = ['After', 'all', 'is', 'said', 'and', 'done', ',',
...           'more', 'is', 'said', 'than', 'done', '.']
>>> for word in saying:
...     print word, '(' + str(len(word)) + '),',
After (5), all (3), is (2), said (4), and (3), done (4), , (1), more (4), is
(2), said (4), than (4), done (4), . (1),
```

Possiamo occuparci dell'ordinazione della linea con l'aiuto del modulo Python `textwrap`. Per la massima chiarezza separereremo ogni passo nella sua riga:

```
>>> from textwrap import fill
>>> format = '%s (%d),'
>>> pieces = [format % (word, len(word)) for word in saying]
>>> output = ' '.join(pieces)
>>> wrapped = fill(output)
>>> print wrapped
After (5), all (3), is (2), said (4), and (3), done (4), , (1), more
(4), is (2), said (4), than (4), done (4), . (1),
```

Notare che c'è una interruzione di riga tra `more` e il numero che lo segue. Se vogliamo evitare ciò. Dovremmo ridefinire la stringa di formattazione in modo che non contenga spazi, es.`'%s_(%d),'` , poi invece di stampare il valore di `wrapped` , possiamo stampare `wrapped.replace('_', ' ')` .

3.10 Riassunto

- In questo libro abbiamo visto un testo come una lista di parole. Un "testo semplice" è potenzialmente una lunga stringa che contiene parole e spazi bianchi che definiscono la formattazione, ed è il modo in cui noi solitamente immagazziniamo e visualizziamo un testo.
- Una stringa è specificata in Python usando la singola o doppia virgoletta: `'Monty Python'` , `"Monty Python"`.
- I caratteri di una stringa sono accessibili usando degli indici, a partire da zero: `'Monty Python'[0]` ci dà il valore `M` . La lunghezza di una stringa è trovata usando `len()`.
- Le sottostringhe sono accessibili usando la notazione slice: `'Monty Python'[1:5]` dà il valore `onty` . Se l'indice di partenza è omesso, la sottostringa comincia all'inizio della stringa; se l'indice finale è omesso, la slice continua fino alla fine della stringa.
- Le stringhe possono essere divise in liste: `'Monty Python'.split()` dà `['Monty', 'Python']` . Le liste possono essere unite in stringhe: `'/'.join(['Monty', 'Python'])` dà `'Monty/Python'` .
- Possiamo leggere il testo da un file `f` utilizzando `text = open(f).read()` . Possiamo leggere un testo da un URL `u` usando `text = urlopen(u).read()`. Possiamo iterare al di là delle righe di un file di testo usando `for line in open(f)`.

- I testi trovati sul web possono contenere materiale indesiderato (come titoli, piè di pagina, markup), che devono essere rimossi prima di compiere qualsiasi elaborazione linguistica.
- La tokenizzazione è la segmentazione di un testo in unità di base -o token- come ad esempio parole e segni di punteggiatura. La tokenizzazione basata sugli spazi bianchi è inadeguata per molte applicazioni perché unisce parole e punteggiatura insieme. NLTK fornisce un tokenizzatore prodotto in serie `nltk.word_tokenize()`.
- La lemmatizzazione è un processo che mappa le varie forme di una parola (come ad esempio *appeared, appears*) fino alla forma canonica o citazionale della parola, conosciuta anche come lessema o lemma (es. *appear*).
- Le espressioni regolari sono un metodo potente e flessibile di specificazione dei modelli. Una volta importato il modulo `re`, possiamo usare `re.findall()` per cercare tutte le sottostringhe in una stringa che corrispondono a un modello.
- Se un'espressione regolare include una barra retroversa, bisognerà dire a Python di non prelaborare la stringa, usando una stringa semplice con il prefisso `r` : `r'regexp'` .
- Quando una barra retroversa è usata prima di certi caratteri, es. `\n` , il tutto assume un significato particolare (il carattere di ritorno a capo); comunque, quando la barra retroversa è usata prima di metacaratteri wildcard e operatori, es. `\.`, `\|`, `\$`, questi caratteri *perdono* il loro significato speciale e sono corrisposti letteralmente.
- Una espressione di formattazione di stringa `template % arg_tuple` consiste di una stringa di formato `template` che contiene specificatori di conversione come `%-6s` e `%0.2d` .

3.11 Ulteriori letture

Materiale in più per questo capitolo lo si può trovare al suto http://www.nltk.org/ , inclusi i link per risorse disponibili gratuitamente sul web. Ricordate di consultare il materiale di riferimento Python sul sito http://docs.python.org/ . (Per esempio, questa documentazione copre la "Universal newline support", che spiega come lavorare con le diverse conversioni di ritorno a capo usato dai diversi sistemi operativi.)

Per più esempi di elaborazione delle parole con NLTK, vedere la tokenizzazione, lo stemming e il corpus HOWTO al sito http://www.nltk.org/howto. Il capitolo 2 e il Capitolo 3 di (Jurafsky & Martin, 2008) contiene più materiale avanzato sulle espressioni regolari e sulla morfologia. Per una discussione più vasta sull'elaborazione dei testi con Python vedere (Mertz, 2003). Per maggiori informazioni riguardo alla normalizzazione delle parole non-standard vedere (Sproat et al, 2001).

Ci sono molte altre referenze per le espressioni regolari, sia pratiche che teoriche. Per un tutorial introduttivo su come usare le espressioni regolari, vedere *Regular Expression HOWTO* di Kuchling's, (Friedl, 2002). Altre presentazioni includono la Sezione 2.1 di (Jurafsky & Martin, 2008), e il Capitolo 3 di (Mertz, 2003).

Ci son molte risorse online per Unicode. Discussioni utili di strutture Python per la manipolazione Unicode sono:

- PEP-100 http://www.python.org/dev/peps/pep-0100/
- Jason Orendorff, *Unicode for Programmers*, http://www.jorendorff.com/articles/unicode/
- A. M. Kuchling, *Unicode HOWTO*, http://www.amk.ca/python/howto/unicode
- Frederik Lundh, *Python Unicode Objects*, http://effbot.org/zone/unicode-objects.htm

- Joel Spolsky, *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*,http://www.joelonsoftware.com/articles/Unicode.html

I problemi di tokenizzazione del testo in Cinese è il punto principale di SIGHAN, l'ACL Special Interest Group on Chinese Language Processing `http://sighan.org/`. Il nostro metodo di segmentazione dei testi in Inglese segue (Brent, 1995); questo lavoro ricade nell'area dell'acquisizione del linguaggio (Niyogi, 2006).

Le collocazioni sono un caso speciale di espressioni multi parola. Una **espressione multi parola** è una piccola frase il cui significato e altre proprietà non possono essere suggerite solamente dalle parole, es. *part of speech* (Baldwin & Kim, 2010).

La ricottura simulata è un eurisma per cercare una buona approssimazione al valore ottimale di una funzione in un vasto, discreto spazio di ricerca, basato su un'analogia con la ricottura in metallurgia. La tecnica è descritta in molti testi sull'Intelligenza Artificiale.

L'approccio alla scoperta degli iponimi nel testo usando modelli di ricerca come *x and other ys* è descritto da (Hearst, 1992).

4 Scrittura di programmi strutturati

A questo punto si avrà un senso di capacità del linguaggio di programmazione Python per l'elaborazione del linguaggio naturale. Tuttavia, se siete nuovi a Python o alla programmazione, si può ancora lottare con Python e non sentire come se sono in pieno controllo ancora. In questo capitolo si affronteranno le seguenti domande:

1. Come si possono scrivere programmi, ben strutturati e leggibili che tu e gli altri sarete in grado di riutilizzare facilmente?
2. Come si costruiscono i blocchi fondamentali del lavoro , come sequenze , funzioni e mansioni?
3. Quali sono alcuni dei problemi con la programmazione Python e come possiamo evitarli?

Lungo il percorso, potrete consolidare la vostra conoscenza sui concetti fondamentali della programmazione, ulteriori informazioni sull'utilizzo di funzionalità del linguaggio Python in modo naturale e conciso, e imparare alcune tecniche utili nella visualizzazione di dati in linguaggio naturale. Come prima, questo capitolo contiene molti esempi ed esercizi (e come prima, alcuni esercizi introducono nuovi materiali). I nuovi lettori della programmazione dovrebbero lavorare tramite loro attentamente e consultare altre introduzioni di programmazione, se necessario; i programmatori esperti possono rapidamente scorrere questo capitolo.

Negli altri capitoli di questo libro, abbiamo organizzato i concetti di programmazione come dettato dalle esigenze di NLP. Qui si ritorna ad un approccio più convenzionale dove il materiale è più strettamente legato alla struttura del linguaggio di programmazione. Non c'è spazio per una presentazione completa del linguaggio, quindi ci basta concentrarsi sui concetti del linguaggio e sugli stili che sono più importanti per la NLP.

4.1 Tornare alle basi

Attribuzione

L' Attribuzione sembra essere il concetto di programmazione più elementare che non merita un discorso a parte. Tuttavia qui, ci sono alcune sottigliezze sorprendenti . Si consideri il seguente frammento di codice:

```
>>> Foo = 'Monty'
>>> Bar = foo
>>> Foo = 'Python'
>>> bar
'Monty'
```

Questo funziona esattamente come previsto. Quando si scrive bar = foo nel codice precedente, il valore di foo ('Monty' la stringa) viene assegnato alla battuta. Cioè, bar è una copia di foo

così quando sovrascriviamo foo con 'Python' una nuova stringa on line, il valore della barra non è condizionante.

Tuttavia le istruzioni di attribuzione non sempre implicano la realizzazione di copie in questo modo. L' Assegnazione copia sempre il valore di un'espressione ma il valore non è sempre quello che ci si potrebbe aspettare che sia. In particolare il "valore" di una struttura oggetto come una lista è in realtà solo un riferimento all'oggetto. Nel seguente esempio si assegna la relazione di foo alla nuova variabile bar. Ora, quando si modifica qualcosa dentro foo on line possiamo vedere che i contenuti di bar sono stati cambiati.

```
>>> Foo = ['Monty', 'Python']
>>> Bar = foo
>>> Foo [1] = 'Bodkin'
>>> bar
['Monty', 'Bodkin']
```

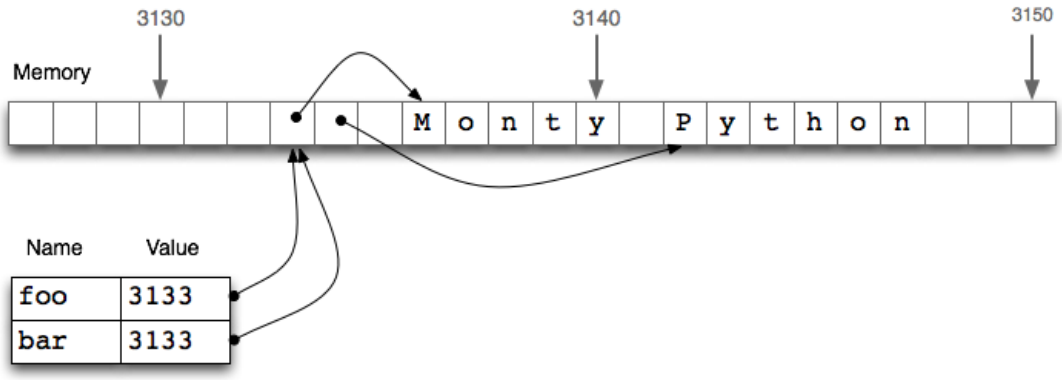


Figura 4.1: Lista degli incarichi e della memoria del computer: Due oggetti dell'elenco foo e bar fanno riferimento alla stessa posizione nella memoria del computer, l'aggiornamento foo modificherà anche bar, e viceversa.

la linea di bar = foo non copia il contenuto della variabile, solo il suo "oggetto di riferimento". Per capire cosa sta succedendo qui, abbiamo bisogno di sapere in che modo gli elenchi vengono memorizzati nella memoria del computer. In 4.1, vediamo che una lista foo è un riferimento a un oggetto archiviato nella posizione 3133 (che di per sé è una serie di puntatori

a stringhe che tengono altre località). Quando si assegna bar = foo, è solo il riferimento all'oggetto 3133 che viene copiato. Questo comportamento si estende ad altri aspetti del linguaggio, come ad esempio il passaggio dei parametri (4.4).

Facciamo ancora un po' sperimentazione, creando un vuoto variabile che contiene la lista vuota, poi utilizzandolo tre volte nella riga successiva.

```
>>> Vuoto = []
>>> Nested = [vuoto, vuoto, vuoto]
>>> nested
[[], [], []]
>>> nested [1]. Append ('Python')
>>> nested
[['Python'], ['Python'], ['Python']]
```

Osservare che cambiando una delle voci all'interno del nostro gruppo di liste cambiano tutti. Questo perché ciascuno dei tre elementi è in realtà solo un riferimento e medesimo elenco in memoria.

Nota

Il vostro turno: Utilizzare la moltiplicazione per creare una lista di liste: nested = [[]] * 3. Ora modificare uno degli elementi della lista, e osservare che tutti gli elementi sono cambiati. Utilizzare id id () per scoprire il codice numerico di qualsiasi oggetto, e verificare che id (nested [0]), id (nested [1]), e id (nested [2]) sono tutti uguali.

Ora, si noti che quando si assegna un nuovo valore a uno degli elementi della lista, esso non si trasmette agli altri:

```
>>> Nested = [[]] * 3
>>> nested [1]. Append ('Python')
>>> nested [1] = ['Monty']
>>> nested
[['Python'], ['Monty'], ['Python']]
```

Abbiamo iniziato con una lista che contiene tre riferimenti ad una singola lista vuota di oggetti. Poi abbiamo modificato l'oggetto aggiungendo 'Python' ad essa, il risultato è un elenco contenente tre riferimenti a un oggetto unico dell'elenco ['Python']. Successivamente,

abbiamo sovrascritto uno di quei riferimenti, con una relazione ad un nuovo oggetto ['Monty']. Quest'ultima operazione modifica uno dei tre oggetti di riferimento all'interno dell' elenco. Tuttavia, il ['Python'] oggetto non è stato modificato, ed è ancora riferimento da due posti nel nostro elenco di liste. E 'fondamentale per valutare questa differenza tra modifica di un oggetto per mezzo di un riferimento a un oggetto, e sovrascrivere un riferimento all'oggetto.

Nota

Importante: Per copiare gli elementi da un elenco foo in una nuova lista bar, è possibile scrivere bar = foo [:]. Questo copia gli oggetti di riferimento all'interno della lista. Per copiare una struttura senza copiare gli oggetti di riferimento, utilizzare copy.deepcopy ().

Uguaglianza

Python fornisce due modi per controllare che un paio di elementi sono gli stessi. Il test is operatore per l'identità dell'oggetto. Possiamo utilizzarlo per verificare le nostre osservazioni precedenti sugli oggetti. Per prima cosa creiamo una lista contenente diverse copie dello stesso oggetto, e dimostrare che non sono solo identici secondo ==, ma anche che sono uno e lo stesso oggetto:

```
>>> Size = 5
>>> Python ['Python']
>>> Snake_nest [Python] * dimensioni

>>> Snake_nest [0] == snake_nest [1] == snake_nest [2] == snake_nest [3] == snake_nest [4]
true
>>> Snake_nest [0] is snake_nest [1] is snake_nest [2] is snake_nest [3] is snake_nest [4]
true
```

Ora mettiamo un nuovo nuovo in questo elenco. Si può facilmente dimostrare che gli oggetti non sono tutti uguali:

```
>>> Import random
>>> Position = random.choice (range (size))
>>> Snake_nest [position] ['Python']
>>> snake_nest
[['Python'], ['Python'], ['Python'], ['Python'], ['Python']]
>>> Snake_nest [0] == snake_nest [1] == snake_nest [2] == snake_nest [3] == snake_nest [4]
true
>>> Snake_nest [0] is snake_nest [1] is snake_nest [2] is snake_nest [3] is snake_nest [4]
False
```

È possibile effettuare diversi test a coppie per scoprire quale posizione contiene l'intruso, ma la funzione id () rende il rilevamento più semplice:

```
>>> [Id (snake) for snake in snake_nest]
[513528, 533168, 513528, 513528, 513528]
```

Questo rivela che la seconda voce della lista ha un identificatore diverso. Se si tenta l'esecuzione di questo frammento di codice da soli, aspettiamo di vedere diversi numeri nella lista risultante, e anche l'intruso potrebbe essere in una posizione differente.

Avere due tipi di uguaglianza potrebbe sembrare strano. Tuttavia, in realtà è solo il tipo di distinzione di token, familiare dal linguaggio naturale, qui mostrato in un linguaggio di programmazione.

Condizionali

Nella condizione del comando if, una stringa non vuota o una lista viene valutata come vera, mentre una stringa vuota o una lista viene valutata come falsa.

```
>>> Mixed ['cat','', ['dog'], []]
>>> for element in mixed:
... if element:
... print element
...
cat
['dog']
```

Cioè, non c'è bisogno di dire if len (element)> 0: nella condizione.

Qual è la differenza tra l'utilizzo di if ... elif anziché utilizzare un paio di istruzioni if in una riga? Beh, si consideri la seguente situazione:

```
>>> Animals ['cat', 'dog']
>>> if 'cat' in animals:
... print 1
... 'elif 'dog' in animals:
... print 2
...
1
```

Dal momento che la clausola if dell'istruzione è soddisfatta, Python non cerca di valutare la

clausola elif, quindi non si arriverà mai a stampare 2. Al contrario, se abbiamo sostituito elif con un if, allora potremmo stampare sia 1 che 2. Quindi, una clausola elif potenzialmente ci dà più informazioni di un vuoto punto if , quando risulta vera, ci dice non solo che la condizione è soddisfatta, ma anche che la condizione del principale punto if non è stata soddisfatta.

Le funzioni all () e any () possono essere applicate ad una lista (o altre sequenze) per verificare se tutti o alcuni elementi soddisfino alcune condizioni:

```
>>> sent = ['No', 'good', 'fish', 'goes', 'anywhere', 'without', 'a', 'porpoise', '.']
>>> all (len (w)> 4 for w in sent)
False
>>> any (len (w)> 4 for w in sent)
true
```

4.2 Sequenze

Finora, abbiamo visto due tipi di sequenze oggetto: stringhe e liste. Un altro tipo di sequenza è chiamata tuple. I tuples sono formati con l'operatore virgola 1, e in genere racchiusi utilizzando le parentesi. Le Abbiamo effettivamente viste nei capitoli precedenti, e talvolta indicandole come "coppia", in quanto vi erano sempre due membri. Tuttavia, i tuple possono avere qualsiasi numero di membri. Come stringhe e liste, i tuples possono essere indicizzati 2 e tagliati 3, e hanno una lunghezza 4.

```
>>> T = 'walk', 'fem', 3 1
>>> t
('walk', 'fem', 3)
>>> T [0] 2
'a piedi'
>>> T [1:] 3
('fem', 3)
>>> Len (t) 4
3
```

Attenzione!

I tuples sono costruiti utilizzando l'operatore virgola. Le parentesi sono una caratteristica più generale di sintassi di Python, progettato per il raggruppamento. Una tupla contenente il singolo elemento 'Snark' è definito con l'aggiunta di una virgola finale, in questo modo: "

'Snark',''. Il tuple vuoto è un caso particolare, ed è definito utilizzando le parentesi vuote ().

Mettiamo a confronto le stringhe, liste e tuple direttamente ed eseguire l'indicizzazione, dividendo in parti, e effettuando l'operazione lunghezza su ogni tipo:

```
>>> raw = 'I turned off the spectroroute'
>>> text = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> pair = (6, 'turned')
>>> raw[2], text[3], pair[1]
('t', 'the', 'turned')
>>> raw[-3:], text[-3:], pair[-3:]
('ute', ['off', 'the', 'spectroroute'], (6, 'turned'))
>>> len(raw), len(text), len(pair)
(29, 5, 2)
```

Si noti che in questo esempio di codice abbiamo calcolato i valori multipli su una singola riga, separati da virgole. Queste espressioni separate da virgole sono in realtà i tuples - Python ci permette di omettere le parentesi che racchiudono i tuples se non c'è ambiguità. Quando si stampa un tuple, le parentesi vengono sempre visualizzate. Utilizzando i tuples in questo modo, stiamo implicitamente aggregando oggetti insieme.

Nota

Il vostro turno: Definire un set, ad esempio utilizzando il set (test) e vedere cosa succede quando si converte in un elenco o si ripete più dei suoi membri.

Operando su Tipi sequenza

Possiamo eseguire un'iterazione su elementi in una sequenza s in una varietà di modi utili, come mostrato in 4.1.

Python Expression	**Commento**
per la voce in s	scorrere gli elementi di s
per la voce in ordinata (s) 4	scorrere gli elementi di s per
per la voce nel set (s) 4	si ripete più elementi unici di s

per la voce in inversa (s) 4 si ripete più elementi di s in senso inverso
per elemento in serie (s) 4. differenza (t) non iterare sugli elementi di s in t

si ripete più elementi di s in ordine casuale
Tabella 4.1:

per la voce in random.shuffle (s) Vari modi per eseguire un'iterazione su sequenze

Le funzioni illustrate in sequenza 4,1 possono essere combinate in vari modi, per esempio, per ottenere elementi unici di s filtrate per inverso, utilizzare reversed (sorted (set (s))).

Siamo in grado di convertire tra questi tipi di sequenza. Ad esempio, tuple (s) converte qualsiasi tipo di sequenza in un tuple, e list (s) 4 converte qualsiasi tipo di sequenza in un elenco. Siamo in grado di convertire una lista di stringhe a una singola stringa utilizzando il join () la funzione, ad esempio ':'. join (words).

Alcuni altri oggetti, ad esempio un FreqDist, possono essere convertiti in una sequenza (usando list ()) e iterazione supporto, ad esempio:

```
>>> raw = 'Red lorry, yellow lorry, red lorry, yellow lorry.'
>>> text = nltk.word_tokenize(raw)
>>> fdist = nltk.FreqDist(text)
>>> list(fdist)
['lorry', ',', 'yellow', '.', 'Red', 'red']
>>> for key in fdist:
...     print fdist[key],
...
4 3 2 1 1 1
```

Nel prossimo esempio, si utilizzano i tuples per riorganizzare i contenuti della nostra lista. (Possiamo omettere le parentesi, perché la virgola ha maggiore precedenza di assegnazione.)

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> words[2], words[3], words[4] = words[3], words[4], words[2]
>>> words
```

['I', 'turned', 'the', 'spectroroute', 'off']

Questo è un modo idiomatico e leggibile per spostare gli elementi all'interno di un elenco. È equivalente al seguente modo tradizionale di fare questi compiti non utilizzando i tuples (notare che questo metodo richiede una variabile temporanea tmp) .

>>> tmp = words[2]

>>> words[2] = words[3]

>>> words[3] = words[4]

>>> words[4] = tmp

Come abbiamo visto, Python ha funzioni esecutive, come sorted () e reversed () che riorganizzano gli elementi di una sequenza. Ci sono anche funzioni che modificano la struttura di una sequenza e che possono essere utili per l'elaborazione del linguaggio. Così, zip () prende gli elementi di due o più sequenze e "zip" insieme in un unico elenco di tuples. Data una sequenza di s, enumerate (s) restituisce coppie costituite da un indice e la voce di tale indice.

>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']

>>> tags = ['noun', 'verb', 'prep', 'det', 'noun']

>>> zip(words, tags)

[('I', 'noun'), ('turned', 'verb'), ('off', 'prep'),

('the', 'det'), ('spectroroute', 'noun')]

>>> list(enumerate(words))

[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]

Per alcuni compiti NLP è necessario tagliare una sequenza in due o più parti. Per esempio, si potrebbe desiderare di "addestrare" un sistema sul 90% dei dati e testare sul restante 10%. Per fare questo abbiamo deciso la posizione in cui si vogliono tagliare i dati 1, quindi tagliare la sequenza in quella posizione 2.

>>> text = nltk.corpus.nps_chat.words()

>>> cut = int(0.9 * len(text)) 1

```
>>> training_data, test_data = text[:cut], text[cut:]   2

>>> text == training_data + test_data  3

True

>>> len(training_data) / len(test_data)   4

9
```

Si può verificare che nessuno dei dati originali viene perso durante questo processo, né è ripetuto 3. Possiamo anche verificare che il rapporto tra le dimensioni dei due pezzi è quello che si intende 4.

La combinazione di diversi tipi di sequenza

Uniamo la nostra conoscenza di questi tre tipi di sequenza, unitamente alla comprensione dell'elenco, per effettuare l'attività di ordinamento delle parole in una stringa per la loro lunghezza.

```
 >>> words = 'I turned off the spectroroute'.split()

 >>> wordlens = [(len(word), word) for word in words]

>>> wordlens.sort()

>>> ' '.join(w for (_, w) in wordlens)

'I off the turned spectroroute'
```

Ciascuna delle suddette linee di codice contiene una caratteristica significativa. Una stringa semplice, è in realtà un oggetto con metodi definiti su di essa come lo split () . Usiamo una lista di comprensione per costruire una lista di tuples, in cui ogni tuple è composta da un numero (la lunghezza della parola) e la parola, ad esempio, (3, 'the'). Usiamo il metodo sort () per ordinare l'elenco sul posto. Infine, eliminare le informazioni di lunghezza e unire le parole di nuovo in una singola stringa. (la sottolineatura è solo una variabile normale Python, ma possiamo usarla per convenzione di sottolineatura per indicare che non si avvarrà del suo valore.)

Abbiamo iniziato parlando dei punti in comune in questi tipi di sequenze, ma il codice di cui sopra illustra importanti differenze nei loro ruoli. In primo luogo, le stringhe appaiono all'inizio e alla fine: questo è tipico nel contesto in cui il nostro programma è la lettura del testo e la produzione in uscita per noi da leggere. Le liste e le tuple sono usate nel mezzo, ma per scopi diversi. Un elenco è tipicamente una sequenza di oggetti aventi tutti lo stesso tipo, di lunghezza arbitraria. Noi spesso utilizziamo gli elenchi per contenere sequenze di parole. Al

contrario, un tuple è tipicamente un insieme di oggetti di diverso tipo, di lunghezza fissa. Spesso usiamo un tuple per tenere un record, un insieme di campi diversi in materia di qualche entità. Questa distinzione tra l'uso di liste e tuple richiede un po 'per abituarsi, ecco un altro esempio:

```
 >>> lexicon = [
...     ('the', 'det', ['Di:', 'D@']),
...     ('off', 'prep', ['Qf', 'O:f'])
... ]
```

Qui, un lessico è rappresentato come una lista perché è un insieme di oggetti di un singolo tipo - entrate lessicali - no di lunghezza predeterminata. Una singola voce è rappresentata come una tuple perché è un insieme di oggetti con interpretazioni diverse, come la forma ortografica, la parte del discorso, e le pronunce (rappresentati nel supporto informatico http://www.phon alfabeto fonetico SAMPA .ucl.ac.uk / home / SAMPA /) . Si noti che queste pronunce sono memorizzate utilizzando un elenco. (Perché?)

Nota

Un buon modo per decidere quando utilizzare i tuples vs liste è quello di chiedere se l'interpretazione di un elemento dipende dalla sua position. Ad esempio, un token con tag unisce due stringhe aventi diversa interpretazione, e abbiamo scelto di interpretare il primo elemento come il token e la seconda voce come il tag. Così si utilizzano i tuples come questo: ('grail', 'noun'), un tuple nella forma ('noun', 'grail') sarebbe privo di senso dal momento che sarebbe un noun tag parola grail. In contrasto, gli elementi di un testo sono tutti tokens, e la posizione non è significativa. Così utilizziamo gli elenchi come questo: ['venetian', 'blind'], una lista della forma ['blind', 'venetian'] sarebbe altrettanto valida. Il significato linguistico delle parole potrebbe essere diverso, ma l'interpretazione di elementi dell'elenco come tokens è invariato.

La distinzione tra liste e tuples è stata descritta in termini di utilizzo. Tuttavia, vi è una differenza fondamentale: in Python, le liste sono mutabilimentre i tuples sono immutabili. In altre parole, gli elenchi possono essere modificati, mentre i tuples no. Qui ci sono alcune delle

operazioni sulle liste che fanno sul posto modifica dell'elenco.

```
>>> lexicon.sort()
>>> lexicon[1] = ('turned', 'VBD', ['t3:nd', 't3`nd'])
>>> del lexicon[0]
```

Nota

Il vostro turno: Convertire lexicon a un tuple, utilizzando lexicon = tuple (lexicon) , quindi provare ciascuna delle operazioni di cui sopra, per confermare che nessuno di loro è consentito su i tuples.

Generatore di espressioni

Abbiamo fatto uso pesante di comprensione dell'elenco, per l'elaborazione compatta e leggibile dei testi. Ecco un esempio in cui abbiamo tokenize e normalizzato un testo:

```
>>> text = '''"When I use a word," Humpty Dumpty said in rather a scornful tone,
... "it means just what I choose it to mean - neither more nor less."'''
>>> [w.lower() for w in nltk.word_tokenize(text)]
['"', 'when', 'i', 'use', 'a', 'word', ',', '"', 'humpty', 'dumpty', 'said', ...]
```

Supponiamo ora di voler elaborare ulteriormente queste parole. Possiamo farlo inserendo l'espressione di cui sopra all'interno di una chiamata a qualche altra funzione 1, ma Python ci permette di omettere le parentesi 2.

```
>>> max([w.lower() for w in nltk.word_tokenize(text)]) 1
'word'
>>> max(w.lower() for w in nltk.word_tokenize(text)) 2
```

'word'

La seconda riga utilizza un **generatore di espressione**. Questo è più di una convenienza di notazione: in molte situazioni di elaborazione del linguaggio, le espressioni del generatore saranno più efficienti. In 1, l' archiviazione per l'oggetto nell'elenco deve essere assegnato prima che il valore di max () viene calcolato. Se il testo è molto grande, questo potrebbe essere lento. In 2, i dati vengono trasmessi alla funzione chiamante. Dal momento che la funzione chiamante deve semplicemente trovare il massimo valore - la parola che viene recentemente ordinata in maniera lessicografica - è in grado di elaborare il flusso di dati senza dover conservare nulla di più rispetto al valore massimo visto finora.

4.3 Questioni di stile

La programmazione è più un'arte che una scienza. L'indiscussa "bibbia" della programmazione, di 2.500 pagine in più volumi lavoro di Donald Knuth, si chiama The Art of Computer Programming. Molti libri sono stati scritti su Literate Programming, riconoscendo che gli esseri umani e non soltanto i computer, devono leggere e comprendere i programmi. Qui soffermarmi su alcune questioni di stile di programmazione che hanno importanti implicazioni per la leggibilità del codice, ivi compresa l'organizzazione del codice, procedurale vs stile dichiarativo, e l'uso di variabili di ciclo.

Python stile codificato

Durante la scrittura di programmi si fanno molte scelte sottili sui nomi, la spaziatura, commenti, e così via. Quando si osserva un codice scritto da altre persone, le differenze superflue di stile rendono più difficile interpretare il codice. Di conseguenza, i progettisti del linguaggio Python hanno pubblicato una guida di stile per il codice Python, disponibile all'indirizzo http://www.python.org/dev/peps/pep-0008/. Il valore sottostante presentato nel Manuale è la coerenza, allo scopo di massimizzare la leggibilità del codice. Abbiamo brevemente esposto alcune delle sue raccomandazioni chiave qui, e rimandato alla guida completa per una discussione dettagliata con esempi.

Layout di codice dovrebbe usare quattro spazi per ogni livello di rientro. Si dovrebbe fare in modo che quando si scrive codice Python in un file, di evitare le schede per il rientro, in quanto questi possono essere fraintesi dai diversi editor di testo e il rientro può essere confuso. Le linee devono essere inferiori a 80 caratteri; se necessario, è possibile interrompere una riga all'interno di parentesi tonde, quadre, o graffe , perché Python è in grado di rilevare che la linea continua verso la riga successiva, ad esempio:

```
>>> cv_word_pairs = [(cv, w) for w in rotokas_words
...                  for cv in re.findall('[ptksvr][aeiou]', w)]
>>> cfd = nltk.ConditionalFreqDist(
...         (genre, word)
...         for genre in brown.categories()
...         for word in brown.words(categories=genre))
>>> ha_words = ['aaahhhh', 'ah', 'ahah', 'ahahah', 'ahh', 'ahhahahaha',
...           'ahhh', 'ahhhh', 'ahhhhhh', 'ahhhhhhhhhhhhhh', 'ha',
...           'haaa', 'hah', 'haha', 'hahaaa', 'hahah', 'hahaha']
```

Se avete bisogno di spezzare una linea fuori le parentesi tonde, quadre, o graffe , spesso è possibile aggiungere parentesi in più, e si può sempre aggiungere una barra rovesciata alla fine della riga che è interrotta:

```
if (len(syllables) > 4 and len(syllables[2]) == 3 and
...    syllables[2][2] in [aeiou] and syllables[2][3] == syllables[1][3]):
...     process(syllables)
>>> if len(syllables) > 4 and len(syllables[2]) == 3 and \
...    syllables[2][2] in [aeiou] and syllables[2][3] == syllables[1][3]:
...     process(syllables)
```

Nota

Digitare spazi al posto delle schede diventa presto un lavoro di routine. Molti editor di programmazione hanno costruito un supporto integrato per Python, ed è possibile far rientrare automaticamente il codice ed evidenziare eventuali errori di sintassi (compresi errori di rientro). Per un

elenco di editors che riconoscono Python, vedere http://wiki.python.org/moin/PythonEditors.

Procedura vs stile dichiarativo

Abbiamo appena visto come la stessa funzione può essere eseguita in vari modi, con implicazioni per l'efficienza. Un altro fattore che influenza lo sviluppo del programma è lo stile di programmazione. Si consideri il seguente programma per calcolare la lunghezza media delle parole del Brown Corpus:

```
>>> tokens = nltk.corpus.brown.words(categories='news')
>>> count = 0
>>> total = 0
>>> for token in tokens:
...     count += 1
...     total += len(token)
>>> print total / count
4.2765382469
```

In questo programma si utilizza la variabile count per tenere traccia del numero dei tokens visti, e total per memorizzare la lunghezza combinata di tutte le parole. Questo è un basso livello di stile, non lontano dal codice macchina, le operazioni primitive eseguite dalla CPU del computer. Le due variabili sono proprio come i registri di CPU, sommano i valori in molti stadi intermedi, valori che sono privi di significato fino alla fine. Diciamo che questo programma è scritto in uno stile procedurale, dettando passo per passo le operazioni della macchina. Si consideri ora il seguente programma che calcola la stessa cosa:

```
>>> total = sum(len(t) for t in tokens)
>>> print total / len(tokens)
4.2765382469
```

La prima riga utilizza un generatore di espressione per sommare le lunghezze del token, mentre la seconda riga calcola la media come prima. Ogni riga di codice esegue una completa,

significativa attività, che può essere compresa in termini di proprietà di alto livello come: "total è la somma delle lunghezze dei tokens". Dettagli di esecuzione sono lasciati per l'interprete di Python. Il secondo programma utilizza una funzione predefinita, e costituisce la programmazione ad un livello più astratto, il codice risultante è più dichiarativo. Vediamo un esempio estremo:

```
>>> word_list = []
>>> len_word_list = len(word_list)
>>> i = 0
>>> while i < len(tokens):
...     j = 0
...     while j < len_word_list and word_list[j] < tokens[i]:
...         j += 1
...     if j == 0 or tokens[i] != word_list[j]:
...         word_list.insert(j, tokens[i])
...         len_word_list += 1
...     i += 1
```

La versione dichiarativa equivalente utilizza familiari funzioni incorporate, e il suo scopo è immediatamente riconoscibile:

```
>>> word_list = sorted(set(tokens))
```

Un altro caso in cui un contatore di ciclo sembra essere necessario per la stampa rappresenta un contatore a ciascuna linea di uscita. Al contrario, possiamo usare enumerate (), che elabora una sequenza s e produce un tuple nella forma (i, s [i]) per ciascuna voce in s, a partire da (0, s [0]). Qui enumerare le chiavi della distribuzione di frequenza, catturare l'intero-stringa coppia nelle variabili rank e word. Stampiamo rank +1 in modo che il conteggio sembra partire da 1, come richiesto quando si produce un elenco di elementi classificati.

```
 >>> fd = nltk.FreqDist(nltk.corpus.brown.words())
>>> cumulative = 0.0
>>> for rank, word in enumerate(fd):
```

```
...     cumulative += fd[word] * 100 / fd.N()
...     print "%3d %6.2f%% %s" % (rank+1, cumulative, word)
...     if cumulative > 25:
...         break
...
  1   5.40% the
  2  10.42% ,
  3  14.67% .
  4  17.78% of
  5  20.19% and
  6  22.40% to
  7  24.29% a
  8  25.97% in
```

A volte è forte la tentazione di utilizzare le variabili di loop per memorizzare un valore minimo o massimo visto finora. Usiamo questo metodo per trovare la parola più lunga in un testo.

```
>>> text = nltk.corpus.gutenberg.words('milton-paradise.txt')
>>> longest = ''
>>> for word in text:
...     if len(word) > len(longest):
...         longest = word
>>> longest
'unextinguishable'
```

Tuttavia, una soluzione più trasparente utilizza due liste di comprensione, aventi forme che dovrebbero essere ormai familiari:

```
>>> maxlen = max(len(word) for word in text)
```

```
>>> [word for word in text if len(word) == maxlen]

['unextinguishable', 'transubstantiate', 'inextinguishable', 'incomprehensible']
```

Si noti che la nostra prima soluzione trova la prima parola con la maggiore lunghezza, mentre la seconda soluzione trova tutte le parole più lunghe (che di solito è quello che noi vogliamo). Anche se c'è una differenza di efficienza teorica tra le due soluzioni, il sovraccarico principale è la lettura dei dati nella memoria principale, una volta che è lì, un secondo passaggio attraverso i dati è effettivamente istantaneo. Abbiamo anche bisogno di bilanciare le nostre preoccupazioni circa l'efficacia del programma con l'efficienza del programmatore. Una soluzione veloce, ma nascosta sarà più difficile da capire e da gestire.

Alcuni usi legittimi per contatori

Ci sono casi in cui si desidera continuare a utilizzare variabili di ciclo di una lista di comprensione. Per esempio, abbiamo bisogno di utilizzare una variabile di ciclo per estrarre successive sovrapposizioni n-grams da un elenco:

```
  >>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']

>>> n = 3

>>> [sent[i:i+n] for i in range(len(sent)-n+1)]

[['The', 'dog', 'gave'],

 ['dog', 'gave', 'John'],

 ['gave', 'John', 'the'],

 ['John', 'the', 'newspaper']]
```

E 'abbastanza difficile da ottenere la portata del diritto variabile del ciclo. Dal momento che questa è un'operazione comune in NLP, NLTK sostiene con funzioni bigrams (text) e trigrams (text), e un generico ngrams (text, n).

Ecco un esempio di come sia possibile utilizzare le variabili di ciclo nella costruzione di strutture multidimensionali. Ad esempio, per costruire una matrice con m righe ed n colonne,

in cui ogni cella è un insieme, potremmo usare un elenco nidificato di comprensione:

```
>>> m, n = 3, 7

>>> array = [[set() for i in range(n)] for j in range(m)]

>>> array[2][5].add('Alice')

>>> pprint.pprint(array)

[[set([]), set([]), set([]), set([]), set([]), set([]), set([])],

 [set([]), set([]), set([]), set([]), set([]), set([]), set([])],

 [set([]), set([]), set([]), set([]), set([]), set(['Alice']), set([])]]
```

Si osservi che le variabili di ciclo i e j non sono utilizzate in tutto l'oggetto risultante, sono solo necessari per una corretta sintassi for. Per fare un altro esempio di questo uso, osservare che l'espressione ['very for i in range (3)] produce un elenco contenente tre istanze di 'very, senza numeri interi in vista.

Si noti che non sarebbe corretto fare questo lavoro con la moltiplicazione, per motivi riguardanti la copia oggetto che sono stati discussi in precedenza in questa sezione.

```
>>> array = [[set()] * n] * m

>>> array[2][5].add(7)

>>> pprint.pprint(array)

[[set([7]), set([7]), set([7]), set([7]), set([7]), set([7]), set([7])],

 [set([7]), set([7]), set([7]), set([7]), set([7]), set([7]), set([7])],

 [set([7]), set([7]), set([7]), set([7]), set([7]), set([7]), set([7])]]
```

L'iterazione è un dispositivo di programmazione importante. Si è tentato di adottare modi di dire da altre lingue. Tuttavia, Python offre alcune alternative eleganti e di facile lettura, come abbiamo visto.

4.4 Funzioni: Le Fondamenta della programmazione strutturata

Funzioni forniscono un modo efficace di confezione e riutilizzo del codice di programma, come già spiegato in 2,3. Per esempio, supponiamo di scoprire che spesso si vuole leggere il testo da un file HTML. Ciò comporta diversi passaggi: aprire il file, leggere dentro, normalizzare gli spazi, e lo stripping del codice HTML. Siamo in grado di raccogliere questi passaggi in una funzione, e dargli un nome, ad esempio get_text (), come indicato al punto 4.2.

```
import re

def get_text(file):
    """Read text from a file, normalizing whitespace and stripping HTML markup."""
    text = open(file).read()
    text = re.sub('\s+', ' ', text)
    text = re.sub(r'<.*?>', ' ', text)
    return text
```

Esempio 4.2 (code_get_text.py): Figura 4.2: leggere testo da un file

Ora, ogni volta che si desidera ottenere ripulito il testo da un file HTML, si può chiamare get_text () con il nome del file come unico argomento. Si restituirà una stringa, e possiamo assegnarlo a una variabile, ad esempio: contents = get_text ("test.html"). Ogni volta che si desidera utilizzare questa serie di passaggi non ci resta che chiamare la funzione.

L'uso delle funzioni ha il vantaggio di risparmiare spazio nel nostro programma. Ancora più importante, la nostra scelta di nome per la funzione che contribuisce a rendere il programma leggibile. Nel caso dell'esempio precedente, ogni volta che il nostro programma deve leggere ripulito il testo da un file non è necessario ingombrare il programma con quattro righe di codice, abbiamo semplicemente bisogno di chiamare get_text (). Questa denominazione contribuisce a fornire un po 'di "interpretazione semantica" - aiuta un lettore del nostro programma a vedere che cosa il programma "significa".

Si noti che la definizione di funzione di cui sopra contiene una stringa. La prima stringa all'interno di una definizione di funzione si chiama docstring. Essa non solo documenta lo scopo della funzione alle persone che leggono il codice, è accessibile a un programmatore che ha caricato il codice da un file:

```
>>> help(get_text)
Help on function get_text:
```

```
get_text(file)
    Read text from a file, normalizing whitespace
    and stripping HTML markup.
```

Abbiamo visto che le funzioni contribuiscono a rendere il nostro lavoro riutilizzabile e leggibile. Inoltre contribuiscono a renderlo più affidabile. Quando si ri-utilizza il codice che è già stato sviluppato e testato, possiamo essere più sicuri che si gestisce una varietà di casi correttamente. Abbiamo anche eliminato il rischio di dimenticarci qualche passo importante, o introdurre un bug. Il programma che chiama la nostra funzione ha anche una maggiore affidabilità. L'autore di questo programma si occupa di un programma più breve, ed i suoi componenti si comportano in modo trasparente.

In sintesi, come suggerisce il nome, una funzione cattura la funzionalità. È un segmento di codice a cui può essere dato un nome significativo e che esegue un compito ben definito. Le funzioni ci permettono di astrarre dai particolari, per vedere la foto più grande, e di programmare in modo più efficace.

Il resto di questa sezione dà un'occhiata più da vicino alle funzioni, esplorando i meccanismi e discutendo dei modi per rendere i programmi più facili da leggere.

Ingressi e uscite di funzione

Passiamo le informazioni alle funzioni utilizzando i parametri di una funzione, l'elenco tra parentesi di variabili e costanti dopo il nome della funzione nella definizione della funzione. Ecco un esempio completo:

```
>>> def repeat(msg, num):
...     return ' '.join([msg] * num)
>>> monty = 'Monty Python'
>>> repeat(monty, 3)
'Monty Python Monty Python Monty Python'
```

Per prima cosa definire la funzione prendendola da due parametri, msg e num. Poi chiamiamo la funzione e trasmetterla a due argomenti, monty e 3; questi argomenti colmano il

"segnaposto" forniti dai parametri e forniscono i valori per le occorrenze di msg e num nel corpo della funzione.

Non è necessario disporre di tutti i parametri, come si vede nell'esempio seguente:

```
>>> def monty():
...     return "Monty Python"
>>> monty()
'Monty Python'
```

Una funzione comunica di solito i suoi risultati al programma chiamante tramite l'istruzione return, come abbiamo appena visto. Per il programma chiamante, sembra come se la chiamata di funzione è stata sostituita con il risultato della funzione, ad esempio:

```
>>> repeat(monty(), 3)
'Monty Python Monty Python Monty Python'
>>> repeat('Monty Python', 3)
'Monty Python Monty Python Monty Python'
```

Una funzione Python non è tenuta ad avere un'istruzione return. Alcune funzioni fanno il loro lavoro come un effetto collaterale, la stampa di un risultato, la modifica di un file o l'aggiornamento del contenuto di un parametro alla funzione (tali funzioni vengono chiamate "procedure" in alcuni altri linguaggi di programmazione).

Consideriamo le seguenti tre funzioni di ordinamento. Il terzo è pericoloso, perché un programmatore potrebbe usarlo senza rendersi conto che aveva modificato il suo ingresso. In generale, le funzioni dovrebbero modificare il contenuto di un parametro (my_sort1 ()), o restituire un valore (my_sort2 ()), non entrambe (my_sort3 ()).

```
>>> def my_sort1(mylist):      # good: modifies its argument, no return value
...     mylist.sort()
>>> def my_sort2(mylist):      # good: doesn't touch its argument, returns value
...     return sorted(mylist)
>>> def my_sort3(mylist):      # bad: modifies its argument and also returns it
```

```
...     mylist.sort()
...     return mylist
```

Passaggio dei parametri

Tornando al punto 4.1 si è visto che l'assegnazione funziona su valori, ma che il valore di un oggetto strutturato è un riferimento a tale oggetto. Lo stesso vale per le funzioni. Python interpreta parametri della funzione come valori (questo è noto come call-by-value). Nel codice seguente, set_up () ha due parametri, entrambi modificati all'interno della funzione. Iniziamo assegnando una stringa vuota a w e una lista vuota a p. Dopo il richiamo della funzione, w è invariato, mentre p è cambiato:

```
 >>> def set_up(word, properties):
...     word = 'lolcat'
...     properties.append('noun')
...     properties = 5
...
>>> w = ''
>>> p = []
>>> set_up(w, p)
>>> w
''
>>> p
['noun']
```

Si noti che w non è stato modificato dalla funzione. Quando abbiamo chiamato set_up (w, p), il valore di w (una stringa vuota) è stato assegnato a una nuova variabile word. All'interno della funzione, il valore di word è stato modificato. Tuttavia, tale cambiamento non si propaga a w. Questo passaggio di parametri è identica alla sequenza di assegnazioni:

```
 >>> w = ''
>>> word = w
```

```
>>> word = 'lolcat'

>>> w

''
```

Diamo un'occhiata a quello che è successo con la lista p. Quando abbiamo chiamato set_up (w, p), il valore di p (un riferimento a una lista vuota) è stato assegnato ad una nuova variabile locale properties, quindi entrambe le variabili ora fanno riferimento alla stessa posizione di memoria. La funzione di modifica properties, e questo cambiamento si riflette anche nel valore di p come abbiamo visto. La funzione inoltre assegna un nuovo valore alla proprietà (il numero 5), questo non ha modificato il contenuto in quella posizione di memoria, ma ha creato una nuova variabile locale. Questo comportamento è come se avessimo fatto la seguente sequenza di assegnazioni:

```
 >>> p = []

>>> properties = p

>>> properties.append('noun')

>>> properties = 5

>>> p

['noun']
```

Così, per capire la chiamata di Python del valore del passaggio di parametri, è sufficiente per capire come funziona l'attribuzione. Ricordate che è possibile utilizzare la funzione id () e is operatore per controllare la vostra comprensione dell'identità dell'oggetto dopo ogni istruzione.

Scopo variabile

Le definizioni di funzione crea un nuovo ambito locale per le variabili. Quando si assegna una nuova variabile all'interno del corpo di una funzione, il nome è definito solo all'interno di tale funzione. Il nome non è visibile all'esterno della funzione, o in altre funzioni. Questo comportamento significa che è possibile scegliere i nomi delle variabili senza preoccuparsi di collisioni con nomi usati nelle definizioni di funzioni.

Quando si fa riferimento a un nome esistente all'interno del corpo di una funzione, l'interprete Python prima tenta di risolvere il nome per quanto riguarda i nomi che sono locali alla funzione. Se non viene trovato, l'interprete controlla se si tratta di un nome globale all'interno del modulo. Infine, se non si riesce, l'interprete controlla se il nome è incorporato in Python.

Questa è la cosiddetta LGB rule di risoluzione dei nomi: locale, globale e poi incorporato.

Attenzione!

Una funzione può creare una nuova variabile globale, utilizzando la dichiarazione global. Tuttavia, tale pratica dovrebbe essere evitata il più possibile. La definizione delle variabili globali all'interno di una funzione introduce dipendenze sul contesto e limita la portabilità (o riutilizzabilità) della funzione. In generale, si consiglia di utilizzare i parametri per gli ingressi delle funzioni e valori di ritorno per le uscite funzione.

Controllare i tipi di Parametro

Python non ci costringe a dichiarare il tipo di una variabile quando si scrive un programma, e questo ci permette di definire le funzioni che sono flessibili sul tipo dei loro argomenti. Ad esempio, un tagger potrebbe aspettare una sequenza di parole, ma non sarebbe importante se questa sequenza è espressa come una lista, una tupla, o un iteratore (un tipo di nuova sequenza che vedremo di seguito).

Tuttavia, spesso si vuole scrivere programmi per il successivo utilizzo da parte di altri, e da programmare secondo uno stile difensivo, fornendo utili avvisi quando le funzioni non sono state invocate correttamente. L'autore del seguente tag () funzione presume che il suo argomento sarebbe sempre una stringa.

```
>>> def tag(word):
...     if word in ['a', 'the', 'all']:
...         return 'det'
...     else:
...         return 'noun'
...
>>> tag('the')
'det'
>>> tag('knight')
'noun'
>>> tag(["'Tis", 'but', 'a', 'scratch'])
'noun'
```

La funzione restituisce valori sensibili per gli argomenti 'the' e 'knight', ma guarda cosa succede quando si passa un elenco - non riesce a lamentarsi, anche se il risultato che viene restituito è chiaramente errato. L'autore di questa funzione potrebbe richiedere ulteriori passi per assicurare che il parametro word del tag () è una stringa. Un approccio più semplice sarebbe quella di controllare il tipo di argomento che utilizza, if not type (word) is str, e se word non è una stringa, il ritorno al semplice valore speciale vuoto di Python, None. Si tratta di un leggero miglioramento, perché la funzione sta verificando il tipo di argomento, e cercando di restituire un "speciale", valore diagnostico per l'ingresso sbagliato. Tuttavia, è anche pericoloso perché il programma chiamante non può rilevare che None è inteso come un valore "speciale", e questo valore di ritorno diagnostico può quindi essere propagato ad altre parti del programma con conseguenze imprevedibili. Questo approccio non riesce anche se la parola è una stringa Unicode, che è di tipo unicode , non str . Ecco una soluzione migliore, utilizzando un assert insieme con il tipo basestring di Python che generalizza sia su unicode e str.

```
>>> def tag(word):
...     assert isinstance(word, basestring), "argument to tag() must be a string"
...     if word in ['a', 'the', 'all']:
...         return 'det'
...     else:
...         return 'noun'
```

Se l'assert fallisce verrà generato un errore che non può essere ignorato dal momento che l'esecuzione del programma si arresta. Inoltre, il messaggio di errore è di facile interpretazione. L'aggiunta di asserzioni ad un programma ti permette di trovare gli errori logici ed è un tipo di programmazione difensiva. Un approccio fondamentale è quello di documentare i parametri alle funzioni mediante docstring come descritto più avanti in questa sezione.

Decomposizione funzionale

I programmi ben strutturati di solito fanno largo uso di funzioni. Quando un blocco di codice di programma si allunga di 10-20 righe, è un grande aiuto per la leggibilità se il codice è suddiviso in una o più funzioni, ognuna con uno scopo chiaro. Questo è analogo al modo in cui è diviso un ottimo articolo nei paragrafi, ognuno esprimendo un idea principale.

Funzioni di fornire un tipo importante di astrazione. Esse ci permettono di raggruppare più azioni in un unico, un'azione complessa, e associare un nome ad esso. (Confronta questo con il nostro modo di combinare le azioni di andare a riportare in un'unica azione più complessa l'operazione di recupero) Quando si utilizzano le funzioni, il programma principale può essere scritto ad un livello di astrazione più alto, rendendo la sua struttura trasparente, ad esempio,

```
>>> data = load_corpus()
>>> results = analyze(data)
>>> present(results)
```

L'uso appropriato di funzioni rende i programmi più leggibili e gestibili. Inoltre, diventa possibile reimplementare una funzione - sostituire il corpo della funzione con un codice più efficiente - senza dover essere interessato con il resto del programma.

Si consideri la funzione freq_words al punto 4.3. Essa aggiorna il contenuto di una distribuzione di frequenza che viene passato come parametro, e stampa anche un elenco delle n parole più frequenti.

```
def freq_words(url, freqdist, n):
    text = nltk.clean_url(url)
    for word in nltk.word_tokenize(text):
        freqdist.inc(word.lower())
    print freqdist.keys()[:n]
```

```
>>> constitution = "http://www.archives.gov/national-archives-experience" \
...              "/charters/constitution_transcript.html"
>>> fd = nltk.FreqDist()
>>> freq_words(constitution, fd, 20)
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

Esempio 4.3 (code_freq_words1.py): Figura 4.3: mal concepita funzione per calcolare Parole frequenti

Questa funzione ha un numero di problemi. La funzione ha due effetti collaterali: si modifica il contenuto del suo secondo parametro, e stampa una selezione dei risultati che ha calcolato. La

funzione sarebbe più facile da capire e da riutilizzare altrove, se si inizializza il FreqDist () dell'oggetto all'interno della funzione (nello stesso luogo esso è popolato), e se abbiamo spostato la selezione e la visualizzazione dei risultati al programma chiamante. In 4.4 abbiamo effettuato il refactoring di questa funzione, e semplificato l'interfaccia, fornendo un unico parametro url.

```
def freq_words(url):
freqdist = nltk.FreqDist()
text = nltk.clean_url(url)
for word in nltk.word_tokenize(text):
    freqdist.inc(word.lower())
    return freqdist
```

```
>>> fd = freq_words(constitution)
>>> print fd.keys()[:20]
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

Esempio 4.4 (code_freq_words2.py): Figura 4.4: Funzione ben progettate per calcolare le parole frequenti

Si noti che ora abbiamo semplificato il lavoro del freq_words al punto che siamo in grado di fare il suo lavoro con tre righe di codice:

```
>>> words = nltk.word_tokenize(nltk.clean_url(constitution))
>>> fd = nltk.FreqDist(word.lower() for word in words)
>>> fd.keys()[:20]
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

Documentare le funzioni

Se abbiamo fatto un buon lavoro al nostro programma di decomposizione in funzioni, allora dovrebbe essere facile da descrivere lo scopo di ogni funzione in un linguaggio semplice, e di fornire questo nel docstring nella parte superiore della definizione della funzione. Questa affermazione non deve spiegare in che modo la funzionalità è implementata, infatti, dovrebbe essere possibile re-implementare la funzione utilizzando un metodo diverso senza cambiare questa dichiarazione.

Per le funzioni più semplici, una riga docstring è di solito sufficiente (vedere 4.2). È necessario fornire una citata tripla stringa contenente una frase completa su una sola riga. Per non banali funzioni, si dovrebbe comunque fornire una sintesi di una frase sulla prima linea, poiché molti indici di elaborazione docstring utilizzano questa stringa. Questo dovrebbe essere seguito da una riga vuota, quindi una descrizione più dettagliata delle funzionalità (vedi http://www.python.org/dev/peps/pep-0257/ per maggiori informazioni nelle convenzioni docstring).

Docstring può includere un doctest blocco, che illustra l'uso della funzione e l'output previsto. Questi possono essere testati automaticamente utilizzando il modulo docutils Python. Docstring dovrebbe documentare il tipo di ciascun parametro alla funzione, e il tipo restituito. Come minimo, che può essere fatto in formato testo. Si noti tuttavia che NLTK utilizza il linguaggio "epytext" marcatura per parametri del documento. Questo formato può essere automaticamente convertito in ricchi programmi di documentazione API (vedi http://www.nltk.org/), e comprende la gestione speciale di alcuni "campi", come @ param che permettono gli ingressi e le uscite delle funzioni chiaramente documentate . 4.5 illustra un docstring completo.

```
def accuracy(reference, test):
  """
  Calculate the fraction of test items that equal the corresponding reference items.

  Given a list of reference values and a corresponding list of test values,
  return the fraction of corresponding values that are equal.
  In particular, return the fraction of indexes
  {0<i<=len(test)} such that C{test[i] == reference[i]}.

    >>> accuracy(['ADJ', 'N', 'V', 'N'], ['N', 'N', 'V', 'ADJ'])
```

```
    0.5


@param reference: An ordered list of reference values.
@type reference: C{list}
@param test: A list of values to compare against the corresponding
    reference values.
@type test: C{list}
@rtype: C{float}
@raise ValueError: If C{reference} and C{length} do not have the
    same length.
"""

if len(reference) != len(test):
    raise ValueError("Lists must have the same length.")
num_correct = 0
for x, y in izip(reference, test):
    if x == y:
        num_correct += 1
return float(num_correct) / len(reference)
```

Esempio 4.5 (code_epytext.py): Figura 4.5: Illustrazione di una elaborazione completa, costituita da un sommario di una linea, una spiegazione più dettagliata, un doctest esempio, e marcatura epytext che determina i parametri, i tipi, il tipo di ritorno, e le eccezioni.

4.5 Fare di più con le funzioni

In questa sezione vengono illustrate le funzionalità più avanzate che si possono preferire di saltare la prima volta attraverso questo capitolo.

Funzioni come argomenti

Fino ad ora gli argomenti che sono passati in funzioni sono stati semplici oggetti come stringhe, o oggetti strutturati come le liste. Python ci permette inoltre di passare una funzione come argomento di un'altra funzione. Ora possiamo estrarre l'operazione, e applicare una diversa operazione sugli stessi dati. Come mostrano gli esempi seguenti, si può passare l'integrazione nella funzione len () o la funzione definita dall'utente last_letter () come argomenti di un'altra funzione:

```
>>> sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
...         'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
>>> def extract_property(prop):
...     return [prop(word) for word in sent]
...
>>> extract_property(len)
[4, 4, 2, 3, 5, 1, 3, 3, 6, 4, 4, 4, 2, 10, 1]
>>> def last_letter(word):
...     return word[-1]
>>> extract_property(last_letter)
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

Gli oggetti len e last_letter possono essere passati in giro come le liste ed i dizionari. Si noti che le parentesi sono utilizzate solo dopo un nome di funzione, se noi invochiamo la funzione; quando stiamo semplicemente trattando la funzione come un oggetto queste sono omesse.

Python ci fornisce un altro modo per definire le funzioni come argomenti per altre funzioni, le cosiddette espressioni lambda. Supponendo che non vi era alcuna necessità di utilizzare sopra la funzione last_letter () in più punti, e quindi non c'è bisogno di dargli un nome. Possiamo equivalentemente scrivere quanto segue:

```
>>> extract_property(lambda w: w[-1])
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

Il prossimo esempio illustra il passaggio di una funzione alla funzione sorted (). Quando chiamiamo quest'ultimo con un singolo argomento (la lista da ordinare), esso utilizza la costruzione in funzione di confronto cmp (). Tuttavia, possiamo fornire la nostra propria funzione di ordinamento, ad esempio, per ordinare la lunghezza decrescente.

```
 >>> sorted(sent)
[',', '.', 'Take', 'and', 'care', 'care', 'of', 'of', 'sense', 'sounds',
'take', 'the', 'the', 'themselves', 'will']
>>> sorted(sent, cmp)
[',', '.', 'Take', 'and', 'care', 'care', 'of', 'of', 'sense', 'sounds',
'take', 'the', 'the', 'themselves', 'will']
>>> sorted(sent, lambda x, y: cmp(len(y), len(x)))
['themselves', 'sounds', 'sense', 'Take', 'care', 'will', 'take', 'care',
'the', 'and', 'the', 'of', 'of', ',', '.']
```

Funzioni accumulative

Queste funzioni cominciano dall'inizializzazione di qualche deposito, e scorrono su input per costruirla, prima di tornare a qualche oggetto finale (una struttura di grandi dimensioni o risultato aggregato). Un modo standard per fare questo è quello di inizializzare una lista vuota, accumulare il materiale, per poi tornare alla lista, come mostrato nella funzione search1 () in 4.6.

```
def search1(substring, words):
   result = []
   for word in words:
      if substring in word:
         result.append(word)
   return result

def search2(substring, words):
```

```
    for word in words:
        if substring in word:
            yield word

print "search1:"
for item in search1('zz', nltk.corpus.brown.words()):
    print item
print "search2:"
for item in search2('zz', nltk.corpus.brown.words()):
    print item
```

Esempio 4.6 (code_search_examples.py): Figura 4.6: Accumulazione di uscita in una lista

La funzione search2 () è un generatore. La prima volta che questa funzione viene chiamata, si ottiene per quanto riguarda l'istruzione yield e le pause. Il programma chiamante ottiene la prima parola e non qualsiasi elaborazione necessaria. Una volta che il programma chiamante è pronto per un'altra parola, l'esecuzione della funzione sarà continuata da dove si era fermata, fino alla prossima volta che incontra una dichiarazione yield. Questo approccio è in genere più efficiente, in quanto la funzione genera solo i dati come richiesto dal programma chiamante, e non ha bisogno di assegnare memoria aggiuntiva per memorizzare l'uscita (cf. la discussione delle espressioni generatore).

Ecco un esempio più sofisticato di un generatore che produce tutte le permutazioni di una lista di parole. Per obbligare la funzione permutations () per generare tutta la sua produzione, ci si avvolge con una chiamata alla list () 1.

```
>>> def permutations(seq):
...     if len(seq) <= 1:
...         yield seq
...     else:
...         for perm in permutations(seq[1:]):
...             for i in range(len(perm)+1):
```

```
...                 yield perm[:i] + seq[0:1] + perm[i:]

...

>>> list(permutations(['police', 'fish', 'buffalo']))

[['police', 'fish', 'buffalo'], ['fish', 'police', 'buffalo'],

 ['fish', 'buffalo', 'police'], ['police', 'buffalo', 'fish'],

 ['buffalo', 'police', 'fish'], ['buffalo', 'fish', 'police']]
```

Nota

La funzione permutations utilizza una tecnica chiamata ricorsione, discusso di seguito in 4.7. La capacità di generare permutazioni di un insieme di parole è utile per creare dati per verificare una grammatica (8).

Funzioni di ordine superiore

Python fornisce alcune funzioni di ordine superiore che sono le caratteristiche standard dei linguaggi di programmazione funzionali come Haskell. Le abbiamo illustrare qui, accanto alle espressioni equivalenti utilizzando la lista di comprensione.

Cominciamo definendo una is_content_word () che verifica se una parola proviene dalla classe libera di parole di contenuto. Usiamo questa funzione come primo parametro del filter (), che applica la funzione di ogni elemento nella sequenza contenuta nel suo secondo parametro, e mantiene solo gli elementi per i quali la funzione restituisce true.

```
 >>> def is_content_word(word):

...     return word.lower() not in ['a', 'of', 'the', 'and', 'will', ',', '.']

>>> sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',

...         'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']

>>> filter(is_content_word, sent)

['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']

>>> [w for w in sent if is_content_word(w)]

['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

Un altro ordine superiore di funzione è map (), che applica una funzione ad ogni elemento in una sequenza. Si tratta di una versione generale della funzione extract_property () che abbiamo visto in 4.5. Ecco un modo semplice per trovare la lunghezza media di una frase nella sezione news del Brown Corpus, seguito da una versione equivalente con lista di comprensione: calcolo:

```
>>> lengths = map(len, nltk.corpus.brown.sents(categories='news'))
>>> sum(lengths) / len(lengths)
21.7508111616
>>> lengths = [len(w) for w in nltk.corpus.brown.sents(categories='news')]
>>> sum(lengths) / len(lengths)
21.7508111616
```

Negli esempi precedenti abbiamo specificato una funzione definita dall'utente is_content_word () e una costruzione nella funzione len (). Possiamo anche fornire un'espressione lambda. Ecco un paio di esempi equivalenti che contano il numero di vocali in ogni parola.

```
>>> map(lambda w: len(filter(lambda c: c.lower() in "aeiou", w)), sent)
[2, 2, 1, 1, 2, 0, 1, 1, 2, 1, 2, 2, 1, 3, 0]
>>> [len([c for c in w if c.lower() in "aeiou"]) for w in sent]
[2, 2, 1, 1, 2, 0, 1, 1, 2, 1, 2, 2, 1, 3, 0]
```

Le soluzioni basate sulla lista di comprensione sono di solito più leggibili rispetto alle soluzioni basate su funzioni di ordine superiore, e abbiamo favorito il primo approccio in questo libro.

Argomenti denominati

Quando ci sono un sacco di parametri è facile confondersi circa l'ordine corretto. Invece si può fare riferimento ai parametri per nome, e persino assegnare loro un valore predefinito nel caso in cui non sono state fornite dal programma chiamante. Ora i parametri possono essere specificati in qualsiasi ordine, e può essere omesso.

```
>>> def repeat(msg='<empty>', num=1):
...     return msg * num
>>> repeat(num=3)
'<empty><empty><empty>'
>>> repeat(msg='Alice')
'Alice'
>>> repeat(num=5, msg='Alice')
'AliceAliceAliceAliceAlice'
```

Questi sono chiamati argomenti chiave. Se mescoliamo questi due tipi di parametri, allora dobbiamo fare in modo che i parametri senza nome precedono quelli citati. Deve essere così, dal momento che i parametri senza nome sono definiti in base alla posizione. Possiamo definire una funzione che prende un numero arbitrario di parametri senza nome e nome, e accedervi tramite un posto nell'elenco degli argomenti * args e un "in-place dictionary" di argomenti chiave ** kwargs. (Dizionari sarà presentato al punto 5.3.)

```
>>> def generic(*args, **kwargs):
...     print args
...     print kwargs
...
>>> generic(1, "African swallow", monty="python")
(1, 'African swallow')
{'monty': 'python'}
```

Quando * args appare come un parametro di funzione, infatti, corrisponde a tutti i parametri senza nome della funzione. Ecco un altro esempio di questo aspetto della sintassi di Python, per la funzione zip () che opera su un numero variabile di argomenti. Useremo la variabile nome * song per dimostrare che non c'è niente di speciale nel nome * args.

```
>>> song = [['four', 'calling', 'birds'],
...         ['three', 'French', 'hens'],
...         ['two', 'turtle', 'doves']]
```

```
>>> zip(song[0], song[1], song[2])

[('four', 'three', 'two'), ('calling', 'French', 'turtle'), ('birds', 'hens', 'doves')]

>>> zip(*song)

[('four', 'three', 'two'), ('calling', 'French', 'turtle'), ('birds', 'hens', 'doves')]
```

Dovrebbe essere chiaro dall'esempio precedente che digitando *song è solo una scorciatoia comoda, e pari a digitando song [0], song [1], song [2].

Ecco un altro esempio di utilizzo di argomenti chiave in una definizione di funzione, insieme a tre modi equivalenti per chiamare la funzione:

```
 >>> def freq_words(file, min=1, num=10):
...     text = open(file).read()
...     tokens = nltk.word_tokenize(text)
...     freqdist = nltk.FreqDist(t for t in tokens if len(t) >= min)
...     return freqdist.keys()[:num]
>>> fw = freq_words('ch01.rst', 4, 10)
>>> fw = freq_words('ch01.rst', min=4, num=10)
>>> fw = freq_words('ch01.rst', num=10, min=4)
```

Un effetto collaterale di argomenti denominati è che essi permettono opzionalità. Così possiamo tralasciare qualsiasi argomento in cui siamo soddisfatti il valore di default: freq_words ('ch01.rst', min = 4), freq_words ('ch01.rst', 4). Un altro uso comune di argomenti facoltativi è di consentire una bandiera. Ecco una nuova versione della stessa funzione che riporta il suo corso, se una bandiera verbose è impostata:

```
 >>> def freq_words(file, min=1, num=10, verbose=False):
...     freqdist = FreqDist()
...     if verbose: print "Opening", file
...     text = open(file).read()
...     if verbose: print "Read in %d characters" % len(file)
```

```
...     for word in nltk.word_tokenize(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...             if verbose and freqdist.N() % 100 == 0: print "."
...     if verbose: print
...     return freqdist.keys()[:num]
```

Attenzione!

Fate attenzione a non utilizzare un oggetto mutabile come valore predefinito di un parametro. Una serie di chiamate alla funzione utilizza lo stesso oggetto, a volte con risultati bizzarri come vedremo nella discussione di debugging di seguito.

4.6 **Sviluppo di Programmi**

La programmazione è una competenza che viene acquisita in diversi anni di esperienza con una varietà di linguaggi di programmazione e dei compiti. Chiave di alto livello di abilità sono la progettazione di algoritmi e la sua manifestazione in programmazione strutturata. Chiave basso livello di abilità includono la familiarità con i costrutti sintattici del linguaggio, e la conoscenza di una varietà di metodi diagnostici per la risoluzione dei problemi di un programma che non presenta il comportamento previsto.

Questa sezione descrive la struttura interna di un modulo di programma e di come organizzare un multi-modulo di programma. Poi descrive i vari tipi di errore che si verificano durante lo sviluppo del programma, cosa si può fare per risolverli e, meglio ancora, e innanzitutto per evitarli.

Struttura di un modulo Python

Lo scopo di un modulo di programma è di portare definizioni logicamente connesse e funzioni insieme al fine di facilitare il riutilizzo e l'astrazione. Moduli Python non sono altro che singoli files. Py. Ad esempio, se si sta lavorando con un formato corpo particolare, le funzioni per leggere e scrivere il formato potrebbero essere tenute insieme. Costanti utilizzate da entrambi i formati, come separatori di campo, o di un EXTN = ". Inf" estensione, potrebbero essere condivisi. Se il formato è stato aggiornato, si sa che un solo file ha bisogno di essere cambiato.

Allo stesso modo, un modulo può contenere il codice per la creazione e la manipolazione di una particolare struttura dati come alberi di sintassi, o il codice per l'esecuzione di una determinata attività indici, come tracciare le statistiche del corpo.

Quando si inizia a scrivere moduli Python, è utile avere alcuni esempi da emulare. È possibile individuare il codice per qualsiasi modulo NLTK sul proprio sistema utilizzando la variabile __ file__, ad esempio:

```
 >>> nltk.metrics.distance.__file__
'/usr/lib/python2.5/site-packages/nltk/metrics/distance.pyc'
```

Ciò restituisce la posizione del compilato. pyc file per il modulo, e probabilmente vedrete una posizione diversa sulla vostra macchina. Il file che hai bisogno di aprire è il corrispondente. py file di origine, e questo sarà nella stessa directory del file. pyc. In alternativa, è possibile visualizzare la versione più recente di questo modulo sul Web all'indirizzo http://code.google.com/p/nltk/source/browse/trunk/nltk/nltk/metrics/distance.py.

Come ogni altro modulo NLTK, distance.py inizia con un gruppo di linee commento dando a una riga il titolo del modulo e identificando gli autori. (Dal momento che il codice viene distribuito, ma include anche l'URL in cui il codice è disponibile, una dichiarazione di copyright, e le informazioni sulla licenza.) La prossima è il livello di modulo docstring, una triplice stringa multilinea che contiene informazioni sul modulo che stamperà tipi di help (nltk.metrics.distance).

```
# Natural Language Toolkit: Distance Metrics
#
# Copyright (C) 2001-2012 NLTK Project
# Author: Edward Loper <edloper@gradient.cis.upenn.edu>
#         Steven Bird <sb@csse.unimelb.edu.au>
#         Tom Lippincott <tom@cs.columbia.edu>
# URL: <http://www.nltk.org/>
# For license information, see LICENSE.TXT
#

"""
```

Distance Metrics.

Compute the distance between two items (usually strings).

As metrics, they must satisfy the following three requirements:

1. d(a, a) = 0

2. d(a, b) >= 0

3. d(a, c) <= d(a, b) + d(b, c)

Seguono tutte le istruzioni di importazione necessari per il modulo, quindi eventuali variabili globali, seguite da una serie di definizioni di funzioni che costituiscono la maggior parte del modulo. Altri moduli si definiscono "classi", il blocco di edificio principale di programmazione orientata agli oggetti, che non rientra nel campo di applicazione di questo libro. (La maggior parte dei moduli NLTK anche una funzione demo () che può essere usata per vedere alcuni esempi del modulo in uso.)

Nota

Alcuni moduli variabili e le funzioni sono utilizzate solo all'interno del modulo. Queste dovrebbero avere nomi che iniziano con un carattere di sottolineatura, ad esempio, _helper (), poiché si nasconde il nome. Se un altro modulo importa questo, usando il linguaggio: from module import *, questi nomi non verranno importati. Facoltativamente, è possibile elencare i nomi accessibili dall'esterno di un modulo utilizzando una speciale variabile incorporata in questo modo: __ all__ = ['edit_distance', 'jaccard_distance'].

Programmi - multi modulo

Alcuni programmi riuniscono una vasta gamma di attività, come ad esempio il caricamento dei dati da un corpus, l'esecuzione di alcune attività di analisi sui dati, quindi visualizzarlo. Potremmo già avere i moduli stabili che si prendono cura di caricamento dei dati e la produzione di visualizzazioni. Il nostro lavoro può comportare la codifica del compito di analisi, e solo invocando le funzioni dei moduli esistenti. Questo scenario è illustrato nella 4.7.

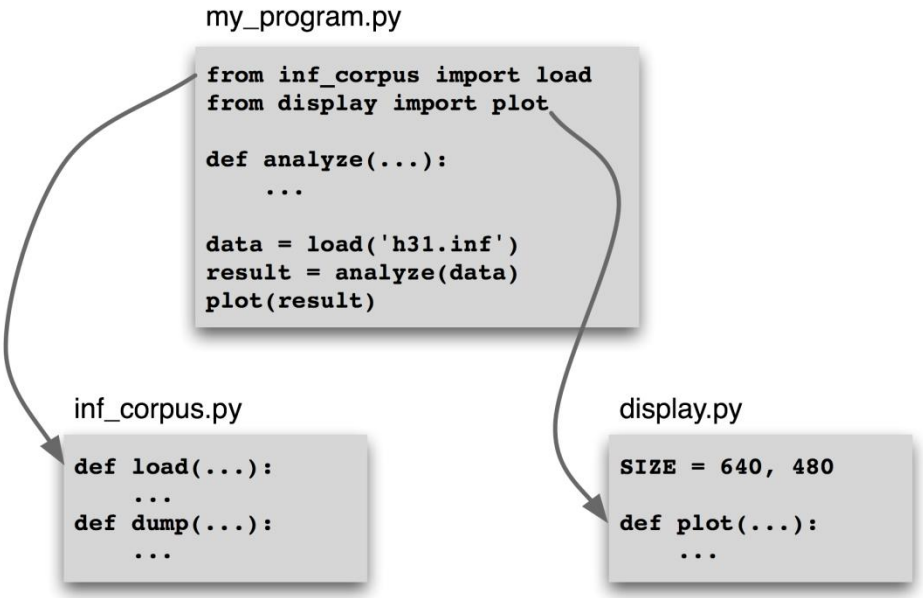


Figura 4.7: Struttura di un multi-modulo di programma: Il programma principale my_program.py importa funzioni da altri due moduli, attività di analisi uniche sono localizzate al programma principale, durante il caricamento comune e la visualizzazione sono tenuti a parte per facilitare il riutilizzo e l'astrazione .

Dividendo il nostro lavoro in diversi moduli e utilizzando l'istruzione import per accedere alle funzioni definite altrove, siamo in grado di mantenere i singoli moduli semplici e di facile manutenzione. Questo approccio comporterà anche una collezione crescente di moduli, e renderà possibile a noi di costruire sofisticati sistemi che comportano una gerarchia di moduli. La progettazione di tali sistemi è un ben complesso lavoro di ingegneria del software, e oltre lo scopo di questo libro.

Fonti di errore

La padronanza di programmazione dipende da una varietà di capacità di problem solving da cui attingere quando il programma non funziona come previsto. Qualcosa di così banale come un errato posto simbolo potrebbe causare il programma di comportarsi in modo molto diverso. Chiamiamo questi "bug" perché sono piccoli in confronto al danno possono causare. Si insinuano nel nostro codice inosservato, ed è solo molto più tardi, quando è in esecuzione il programma su alcuni nuovi dati che la loro presenza viene rilevata. A volte, fissando un bug rivela solo un altro, ed abbiamo la netta impressione che il bug è in movimento. La rassicurazione unica che abbiamo è che i bug sono spontanei e non è colpa del programmatore.

Leggerezza a parte, il debug di codice è difficile perché ci sono tanti modi per essere difettoso. La nostra comprensione dei dati di input, l'algoritmo, o anche il linguaggio di programmazione, può essere esente da colpa. Diamo un'occhiata ad alcuni esempi di ciascuno di questi.

Per primo, i dati di ingresso possono contenere alcuni caratteri imprevisti. Ad esempio, i nomi di synset WordNet hanno la forma tree.n.01, con tre componenti separate utilizzando periodi. Il modulo NLTK WordNet inizialmente scompone questi nomi utilizzando split ('.'). Tuttavia, questo metodo ha rotto quando qualcuno ha cercato di cercare la parola PhD, che ha il nome di ricerca ph.d .. N.01, contenente quattro periodi anziché i previsti due. La soluzione era quella di utilizzare rsplit ('.', 2) per fare al massimo due divisioni, con le istanze più a destra di quel periodo, e lasciando la stringa ph.d. intatta. Anche se diverse persone hanno provato il modulo prima della sua uscita, è stato qualche settimana prima che qualcuno ha rilevato il problema (vedi http://code.google.com/p/nltk/issues/detail?id=297).

In secondo luogo, una funzione in dotazione potrebbe non funzionare nel modo previsto. Ad esempio, durante la prova di interfaccia NLTK di WordNet, uno degli autori notò che nessuna synsets ha avuto contrari definiti, anche se il database sottostante ha fornito una grande quantità di informazioni contrarie. Quello che sembrava un bug nell'interfaccia WordNet si è rivelato un malinteso su WordNet stesso: contrari sono definiti per lemmi, non per synset. L'unico "bug" è stato un malinteso dell'interfaccia (vedi http://code.google.com/p/nltk/issues/detail?id=98).

In terzo luogo, la nostra comprensione della semantica di Python può essere esente da colpa. È facile fare l'ipotesi sbagliata su relativo ambito di due operatori. Ad esempio, "..% S% s% 02d"% "ph.d.", "n", 1 produce una fase di esecuzione TypeError : not enough arguments for format string. Questo perché l'operatore percentuale ha una precedenza maggiore dell'operatore virgola. La correzione consiste nell'aggiungere parentesi per obbligare gli obiettivi necessari. Per fare un altro esempio, supponiamo che stiamo definendo una funzione per raccogliere tutti i token di un testo con una data lunghezza. La funzione ha parametri per il testo e la lunghezza di parola, e un parametro aggiuntivo che permette il valore iniziale del risultato da dare come parametro:

```
>>> def find_words(text, wordlength, result=[]):
...     for word in text:
...         if len(word) == wordlength:
...             result.append(word)
...     return result
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 3)
['omg', 'teh', 'teh', 'mat']
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 2, ['ur'])
['ur', 'on']
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 3)
['omg', 'teh', 'teh', 'mat', 'omg', 'teh', 'teh', 'mat']
```

La prima volta che chiamiamo find_words (), otteniamo tutte le parole di tre lettere come previsto. La seconda volta abbiamo specificato un valore iniziale per il risultato, un solo elemento della lista ['ur'], e come previsto, il risultato ha questa parola insieme alle altre due lettere della parola nel nostro testo. Ora, la prossima volta che noi chiamiamo find_words () si utilizzano gli stessi parametri , ma si ottiene un risultato diverso! Ogni volta che chiamiamo find_words () senza terzo parametro, il risultato si limita ad estendere il risultato della chiamata precedente, piuttosto che iniziare con la lista risultati vuoti, come specificato nella definizione della funzione. Il comportamento del programma non è quello desiderato, perché erroneamente presume che il valore di default è stato creato al momento della chiamata della funzione. Tuttavia, si è creata una sola volta, nel momento che l'interprete Python carica la funzione. Questo oggetto lista è usato quando nessun valore esplicito viene fornito alla funzione.

Tecniche di debug

Dal momento che la maggior parte degli errori di codice deriva dal programmatore facendo ipotesi errate, la prima cosa da fare quando si scopre un bug è quello di verificare le ipotesi. Localizzare il problema con l'aggiunta di istruzioni di stampa del programma, che mostra il valore delle variabili importanti, e mostrando in che misura il programma ha registrato progressi.

Se il programma ha prodotto una "eccezione" - un errore di esecuzione - l'interprete stampa una traccia dello stack, individuando la posizione di esecuzione del programma nel momento dell'errore. Se il programma dipende da dati di input, tentare di ridurre questo alla dimensione più piccola pur continuando a produrre l'errore.

Dopo aver localizzato il problema di una particolare funzione o di una riga di codice, è necessario capire cosa c'è di sbagliato. Spesso è utile per ricreare la situazione utilizzando la riga di comando interattivo. Definire alcune variabili quindi copiare e incollare la riga di codice nella sessione e vedere cosa succede. Controlla la tua comprensione del codice leggendo un po 'di documentazione, esempi di codice e l'esame di altri che pretendono di fare la stessa cosa che si sta cercando di fare. Prova a spiegare il codice a qualcun altro, nel caso in cui è possibile vedere dove le cose vanno male.

Python fornisce un debugger che permette di monitorare l'esecuzione del programma, specificare i numeri di riga in cui l'esecuzione si ferma (ad esempio i punti di interruzione), e passo attraverso le sezioni di codice e controllare il valore delle variabili. È possibile richiamare il debugger per il codice come segue:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.myfunction()')
```

Si presenterà un prompt (Pdb), dove è possibile digitare le istruzioni per il debugger. Digitate help per visualizzare l'elenco completo dei comandi. Digitando passo (o semplicemente s) eseguirà la riga corrente e stop. Se la corrente di linea chiama una funzione, entrerà la funzione e si fermerà alla prima riga. Digitando successivo (o semplicemente n) è simile, ma si ferma l'esecuzione alla riga successiva nella funzione corrente. La rottura (o b) comando può essere utilizzata per creare o elencare i punti di interruzione. Digitare continuare (o c) per continuare l'esecuzione fino al successivo punto di interruzione. Digitare il nome di una variabile per ispezionare il suo valore.

Possiamo usare il debugger Python per individuare il problema nei nostri find_words () la funzione. Ricordate che il problema è sorto per la seconda volta che è stata chiamata la funzione. Inizieremo chiamando la funzione senza utilizzare il debugger 1, utilizzando l'ingresso più piccolo possibile. La seconda volta, noi lo chiamiamo con il debugger 2.

```
 >>> import pdb
>>> find_words(['cat'], 3)  1
['cat']
>>> pdb.run("find_words(['dog'], 3)")  2
> <string>(1)<module>()
(Pdb) step
--Call--
> <stdin>(1)find_words()
(Pdb) args
text = ['dog']
wordlength = 3
result = ['cat']
```

Qui abbiamo digitato solo due comandi nel debugger: il passaggio ci ha portato all'interno della funzione, e args ha mostrato i valori dei suoi argomenti (o parametri). Vediamo subito che il risultato ha un valore iniziale di ['cat'], e non la lista vuota come previsto. Il debugger ci ha aiutato a localizzare il problema, ci spinge a controllare la nostra comprensione delle funzioni di Python.

Programmazione difensiva

Al fine di evitare una parte del dolore di debug, è utile adottare alcune abitudini di programmazione difensive. Invece di scrivere un programma di 20 linee quindi testare, creare il programma ascendente su piccoli pezzi che sono noti per lavorare. Ogni volta che si combinano questi pezzi per fare una unità più grande, verificare con attenzione per vedere che funziona come previsto. Considerare l'aggiunta di statement assert al codice, specificare le proprietà di una variabile, ad esempio assert (isinstance (text, list)). Se il valore della variabile di testo diventa in seguito una stringa quando il codice viene utilizzato in un contesto più ampio, tale da far salire un AssertionError e otterrete la notifica immediata del problema.

Una volta che pensi di aver trovato il bug, visualizzare la soluzione come ipotesi. Cercate di prevedere l'effetto del bugfix prima di ri-eseguire il programma. Se il bug non è stato risolto, non cadere nella trappola di modificare ciecamente il codice, nella speranza che possa magicamente ricominciare a lavorare. Invece, per ogni cambiamento, cercare di esprimere un'ipotesi su ciò che è sbagliato e perché il cambiamento risolverà il problema. Poi annullare la modifica se il problema non è stato risolto.

Quando si sviluppa il programma, estendere le sue funzionalità, e correggere eventuali bug, aiuta a mantenere un insieme di casi di test. Questo è chiamato il test di regressione, poiché lo scopo di rilevare situazioni in cui il codice "regredisce" - dove una modifica al codice ha un indesiderato effetto collaterale di rompere qualcosa che prima funzionava. Python fornisce un quadro semplice di test di regressione nella forma del modulo doctest. Questo modulo cerca un file di codice o di documentazione per blocchi di testo che assomigliano a una sessione interattiva di Python, nella forma che avete già visto molte volte in questo libro. Esegue i comandi Python che trova, e le prove che la loro produzione soddisfa l'output fornito nel file originale. Ogni volta che c'è una mancata corrispondenza, esso segnala i valori previsti ed effettivi. Per maggiori informazioni si prega di consultare la documentazione in doctest http://docs.python.org/library/doctest.html. Oltre al suo valore per i test di regressione, il doctest modulo è utile per garantire che la documentazione del software rimane sincronizzato con il codice.

Forse la strategia più importante di programmazione difensiva è di stabilire in modo chiaro il codice, scegliere la variabile significativa e i nomi delle funzioni, e semplificare il codice, ove possibile dalla decomposizione nelle funzioni e moduli con interfacce ben documentati.

4.7 Algoritmo di design

Questa sezione illustra i concetti più avanzati, che si può preferire di saltare la prima volta che si legge questo capitolo.

Una parte importante di soluzione di problemi algoritmici è la selezione o l'adattamento di un appropriato algoritmo per il problema a portata di mano. A volte ci sono diverse alternative, e la scelta del migliore dipende dalla conoscenza di come ogni alternativa si comporta come la dimensione dei dati cresce. Interi libri sono stati scritti su questo argomento, e abbiamo solo lo spazio per introdurre alcuni concetti fondamentali e approfondire gli approcci che sono i più diffusi nella elaborazione del linguaggio naturale.

La migliore strategia è nota come conosciuta divisione e conquista. Abbiamo attaccato un problema di dimensione n dividendolo in due problemi di dimensione n / 2, risolvere questi problemi, e combinare i risultati in una soluzione del problema originale. Per esempio, supponiamo che abbiamo avuto un mucchio di carte con una sola parola scritta su ogni carta. Potremmo ordinare questo mucchio dividendolo a metà e darlo ad altre due persone per ordinarlo (si potrebbe fare lo stesso, a sua volta). Poi, quando due mucchi ordinati tornano, è un compito facile per unirli in una singola pila ordinata. Vedere 4.8 per un esempio di questo processo.

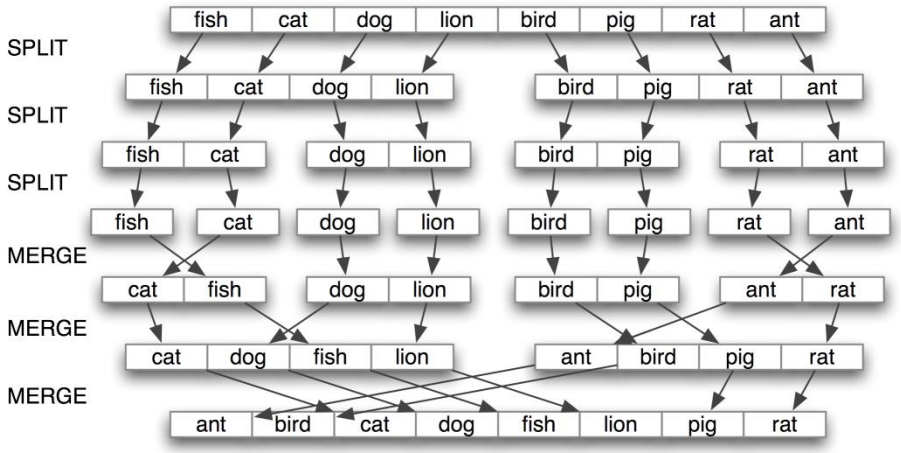


Figura 4.8: Ricerca per Divisione e conquista: per ordinare una matrice abbiamo diviso a metà e ordinato ogni mezzo (ricorsivamente), si fondono ogni metà ordinate di nuovo in un intero elenco (ancora una volta in modo ricorsivo), questo algoritmo è noto come "Merge sort ".

Un altro esempio è il processo di ricerca di una parola in un dizionario. Apriamo il libro da qualche parte intorno alla metà e confrontiamo la nostra parola con la pagina corrente. Se la

sua prima nel dizionario si ripete il processo rispetto al primo semestre, se poi si usa il secondo tempo. Questo metodo di ricerca si chiama ricerca binaria in quanto divide il problema in due ad ogni passo.

In un altro approccio alla progettazione di algoritmi, attacchiamo un problema trasformandolo in un esempio di un problema che già sappiamo come risolvere. Ad esempio, al fine di rilevare voci duplicate in un elenco, si può pre-ordinare l'elenco, quindi eseguire la scansione attraverso di essa una volta per verificare eventuali coppie adiacenti di elementi identici.

Ricorsione

Gli esempi sopra di ordinamento e ricerca hanno una sorprendente proprietà: per risolvere un problema di dimensione n, dobbiamo rompere a metà e poi lavorare su uno o più problemi di dimensione n / 2. Un modo comune per implementare tali metodi utilizza la ricorsione. Definiamo una funzione f che semplifica il problema, e si definisce per risolvere uno o più istanze facili dello stesso problema. Unisce quindi i risultati in una soluzione per il problema originale.

Per esempio, supponiamo di avere un insieme di n parole, e si vuole calcolare quanti modi diversi possono essere combinati per fare una sequenza di parole. Se abbiamo una sola parola (n = 1), non è solo un modo per farlo in una sequenza. Se abbiamo un insieme di due parole, ci sono due modi per mettere in una sequenza. Per tre parole ci sono sei possibilità. In generale, per n parole, ci sono n × n-1 × ... × 2 × 1 modi (cioè il fattoriale di n). Siamo in grado di codificare questo come segue:

```
>>> def factorial1(n):
...     result = 1
...     for i in range(n):
...         result *= (i+1)
...     return result
```

Tuttavia, vi è anche un algoritmo ricorsivo per risolvere questo problema, in base alla seguente osservazione. Supponiamo di avere un modo per costruire tutti gli ordinamenti per n-1 parole distinte. Poi per ogni ordine tale, ci sono n posti dove si può inserire una nuova parola: all'inizio, alla fine o uno qualsiasi degli n-2 confini tra le parole. Così dobbiamo semplicemente moltiplicare il numero di soluzioni trovati per n-1 dal valore di n. Abbiamo anche bisogno del caso base, per dire che se abbiamo una sola parola, c'è solo un ordine. Siamo in grado di codificare questo come segue:

```
>>> def factorial2(n):
...     if n == 1:
...         return 1
...     else:
...         return n * factorial2(n-1)
```

Questi due algoritmi risolvono lo stesso problema. Si utilizza l'iterazione, mentre l'altro usa ricorsione. Siamo in grado di utilizzare la ricorsione per navigare in un oggetto profondamente nidificato come ad esempio la gerarchia hypernym WordNet. Contiamo le dimensioni della gerarchia hypernym radicata in un determinato synset s. Lo faremo, trovando la dimensione di ogni hyponym di s, quindi l'aggiunta di questi insieme (ci sarà anche da aggiungere 1 per lo stesso synset). La seguente funzione size1 () funziona, avviso che il corpo della funzione include una chiamata ricorsiva a size1 ():

```
>>> def size1(s):
...     return 1 + sum(size1(child) for child in s.hyponyms())
```

Possiamo inoltre progettare una soluzione iterativa a questo problema che elabora la gerarchia a strati. Il primo strato è la stessa synset, quindi tutti gli hyponyms del synset, poi tutti gli hyponyms degli hyponyms. Ad ogni iterazione del ciclo si calcola il livello successivo, trovando gli hyponyms di tutto l'ultimo strato. Esso mantiene anche un totale del numero di synsets incontrate finora.

```
>>> def size2(s):
...     layer = [s]
...     total = 0
...     while layer:
...         total += len(layer)
...         layer = [h for c in layer for h in c.hyponyms()]
...     return total
```

Non solo è la soluzione iterativa più lunga, è più difficile da interpretare. Ci costringe a pensare proceduralmente, e tenere traccia di ciò che sta accadendo con il livello e le variabili totali nel tempo. Proviamo a soddisfare entrambe le soluzioni che danno lo stesso risultato. Useremo una nuova forma della dichiarazione di importazione, che ci permette di abbreviare il nome di

WordNet a wn:

```
 >>> from nltk.corpus import wordnet as wn
>>> dog = wn.synset('dog.n.01')
>>> size1(dog)
190
>>> size2(dog)
190
```

Come ultimo esempio di ricorsione, lo si può usare per costruire un oggetto profondamente nidificato. Un trie lettera è una struttura di dati che può essere utilizzata per indicizzare un lessico, una lettera alla volta. (Il nome è basato sul recupero parola). Ad esempio, se trie conteneva un trie lettera, poi trie ['c'] sarebbe un trie piccola che tiene tutte le parole che iniziano con c. 4,9 dimostra il processo ricorsivo di costruzione di un trie, usando dizionari Python (5.3). Per inserire la parola chien (Francese per il cane), abbiamo diviso fuori dal c e ricorsivamente inserire hien nella sub-trie trie ['c']. La ricorsione continua finché non ci sono lettere restanti nella parola, quando si memorizza il valore previsto (in questo caso, la parola cane).

```
 def insert(trie, key, value):
   if key:
      first, rest = key[0], key[1:]
      if first not in trie:
         trie[first] = {}
      insert(trie[first], rest, value)
   else:
      trie['value'] = value
```

```
 >>> trie = {}
>>> insert(trie, 'chat', 'cat')
>>> insert(trie, 'chien', 'dog')
```

```
>>> insert(trie, 'chair', 'flesh')

>>> insert(trie, 'chic', 'stylish')

>>> trie = dict(trie)              # for nicer printing

>>> trie['c']['h']['a']['t']['value']

'cat'

>>> pprint.pprint(trie)

{'c': {'h': {'a': {'t': {'value': 'cat'}},

                {'i': {'r': {'value': 'flesh'}}},

           'i': {'e': {'n': {'value': 'dog'}}}

                {'c': {'value': 'stylish'}}}}}
```

Esempio 4.9 (code_trie.py): Figura 4.9: Costruzione di un Trie lettera: Una funzione ricorsiva che costruisce una struttura nidificata dizionario: ogni livello di nidificazione contiene tutte le parole con un determinato prefisso, e un sub-trie contenente tutte le possibili continuazioni.

Attenzione!

Nonostante la semplicità di programmazione ricorsiva, si tratta di un costo. Ogni volta che viene chiamata una funzione, qualche informazione di stato deve essere spinta su una pila, in modo che una volta che la funzione è completata, l'esecuzione può continuare da dove si è interrotta. Per questo motivo, iterativo le soluzioni sono spesso più efficiente rispetto alle soluzioni ricorsive.

Spazio-**Tempo** Compromessi

Talvolta possiamo velocizzare sensibilmente l' esecuzione di un programma da costruire una struttura ausiliaria di dati, come un indice. La quotazione in 4,10 attua un semplice sistema di recupero di testo per il Movie Reviews Corpus. Per indicizzare la collezione di documenti esso fornisce una ricerca molto più veloce.

```
 def raw(file):
   contents = open(file).read()
   contents = re.sub(r'<.*?>', ' ', contents)
   contents = re.sub('\s+', ' ', contents)
```

```
    return contents


def snippet(doc, term): # buggy
    text = ' '*30 + raw(doc) + ' '*30
    pos = text.index(term)
    return text[pos-30:pos+30]


print "Building Index..."
files = nltk.corpus.movie_reviews.abspaths()
idx = nltk.Index((w, f) for f in files for w in raw(f).split())


query = ''
while query != "quit":
    query = raw_input("query> ")
    if query in idx:
        for doc in idx[query]:
            print snippet(doc, query)
    else:
        print "Not found"
```

Esempio 4.10 (code_search_documents.py): Figura 4.10: Un semplice sistema di Text Retrieval

Un esempio più sottile di uno spazio-tempo compromesso consiste nel sostituire i segni di un corpus con identificatori interi. Creiamo un vocabolario per il corpus, un elenco in cui è memorizzata ogni parola una sola volta, quindi capovolgere questa lista in modo da poter cercare una parola per trovare il suo identificatore. Ogni documento viene pre-elaborato, in modo che un elenco di parole diventa un elenco di numeri interi. Eventuali modelli di linguaggio possono ora lavorare con i numeri interi. Vedere l'elenco di 4.11 per un esempio di come eseguire questa operazione per un corpus tag.

```
def preprocess(tagged_corpus):
   words = set()
   tags = set()
   for sent in tagged_corpus:
      for word, tag in sent:
         words.add(word)
         tags.add(tag)
   wm = dict((w,i) for (i,w) in enumerate(words))
   tm = dict((t,i) for (i,t) in enumerate(tags))
   return [[(wm[w], tm[t]) for (w,t) in sent] for sent in tagged_corpus]
```

Esempio 4.11 (code_strings_to_ints.py): Figura 4.11: pre-elaborazione dei dati del corpus etichetta, la conversione di tutte le parole e tag di interi

Un altro esempio di spazio-tempo compromesso è mantenere un elenco di vocabolario. Se avete bisogno di elaborare un testo di input per verificare che tutte le parole sono in un vocabolario esistente, il vocabolario dovrebbe essere memorizzato come un insieme e non un elenco. Gli elementi di un insieme vengono indicizzati automaticamente, quindi testare l' appartenenza di un insieme grande sarà molto più veloce rispetto a testare appartenenza alla lista corrispondente.

Siamo in grado di verificare questa affermazione utilizzando il modulo timeit. La classe Timer dispone di due parametri, una dichiarazione che viene eseguita più volte, e il codice di installazione che viene eseguito una volta all'inizio. Si simulerà un vocabolario di 100.000 elementi utilizzando un elenco o una serie di numeri interi. La dichiarazione test genera un elemento casuale che ha una probabilità del 50% di essere nel vocabolario.

```
 >>> from timeit import Timer
>>> vocab_size = 100000
>>> setup_list = "import random; vocab = range(%d)" % vocab_size
>>> setup_set = "import random; vocab = set(range(%d))" % vocab_size
>>> statement = "random.randint(0, %d) in vocab" % (vocab_size * 2)
>>> print Timer(statement, setup_list).timeit(1000)
2.78092288971
```

```
>>> print Timer(statement, setup_set).timeit(1000)
0.0037260055542
```

L'esecuzione di 1000 test di appartenenza dell'elenco richiede un totale di 2,8 secondi, mentre le prove equivalenti su un insieme prendono appena 0,0037 secondi, o tre ordini di grandezza più veloce!

Programmazione Dinamica

La programmazione dinamica è una tecnica generale per la progettazione di algoritmi che è ampiamente usato nella elaborazione del linguaggio naturale. 'Programmazione' Il termine è usato in un senso diverso da quello che ci si potrebbe aspettare, a significare la pianificazione o programmazione. Programmazione dinamica viene utilizzata quando un problema di sovrapposizione contiene sotto-problemi. Invece di calcolare soluzioni a questi sotto-problemi più volte, semplicemente li memorizza in una tabella di ricerca. Nella parte restante di questa sezione introdurremo la programmazione dinamica, ma in un contesto un po 'diverso per l'analisi sintattica.

Pingala è stato un autore indiano che visse intorno al 5 ° secolo aC, e scrisse un trattato sulla Sanskrit prosody chiamato Shastra Chandas. Virahanka estese questo lavoro intorno al 6 ° secolo dC, studiando il numero di modi per combinare sillabe corte e lunghe per creare un metro di lunghezza n. Sillabe brevi, segnate S, prendere una unità di lunghezza, mentre sillabe lunghe, contrassegnate dalla lettera L, prendi due. Pingala trovò, per esempio, che ci sono cinque modi per costruire un metro di lunghezza 4: V4 = {LL, SSL, SLS, LSS, SSSS}. Osservate che possiamo dividere V4 in due sottogruppi, quelli che iniziano con L e quelli che iniziano con S, come mostrato in (1).

(1) V4 =

LL, LSS

i.e. L prefixed to each item of V2 = {L, SS}

SSL, SLS, SSSS

i.e. S prefixed to each item of V3 = {SL, LS, SSS}

```
def virahanka1(n):
    if n == 0:
        return [""]
```

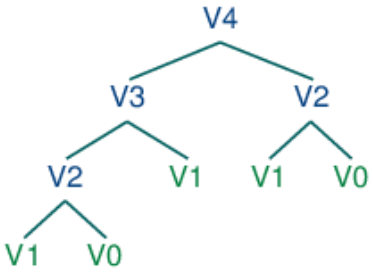
```
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka1(n-1)]
        l = ["L" + prosody for prosody in virahanka1(n-2)]
        return s + l

def virahanka2(n):
    lookup = [[""], ["S"]]
    for i in range(n-1):
        s = ["S" + prosody for prosody in lookup[i+1]]
        l = ["L" + prosody for prosody in lookup[i]]
        lookup.append(s + l)
    return lookup[n]

def virahanka3(n, lookup={0:[""], 1:["S"]}):
    if n not in lookup:
        s = ["S" + prosody for prosody in virahanka3(n-1)]
        l = ["L" + prosody for prosody in virahanka3(n-2)]
        lookup[n] = s + l
    return lookup[n]

from nltk import memoize
@memoize
def virahanka4(n):
    if n == 0:
        return [""]
    elif n == 1:
```

```
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka4(n-1)]
        l = ["L" + prosody for prosody in virahanka4(n-2)]
        return s + l
```

```
 >>> virahanka1(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka2(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka3(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka4(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

Esempio 4.12 (code_virahanka.py): Figura 4.12: Quattro modi per calcolare Sanskrit Meter: (i) iterativo, (ii) bottom-up di programmazione dinamica, (iii) top-down di programmazione dinamica e (iv) memorizzazione incorporata.

Con questa osservazione, possiamo scrivere una piccola funzione ricorsiva chiamata virahanka1 () per calcolare i metri, illustrati in 4.12. Si noti che, per calcolare V4 prima calcoliamo V3 e V2. Ma per calcolare V3, dobbiamo calcolare prima V2 e V1. Questa struttura chiamata è raffigurata in (2).



Come si può vedere, V2 viene calcolata due volte. Questo potrebbe non sembrare un problema significativo, ma risulta essere piuttosto dispendioso come n diventa grande: per calcolare V20 con questa tecnica ricorsiva, si potrebbe calcolare V2 4181 volte, e per il V40 si potrebbe calcolare V2 63.245.986 volte! Un'alternativa migliore è quella di memorizza il valore di V2 in una tabella e guardare in su ogni volta che ne abbiamo bisogno. Lo stesso vale per altri valori, quali V3 e così via. Funzione virahanka2 () implementa un approccio dinamico al problema di programmazione. Funziona attraverso la compilazione di una tabella (chiamata più veloce), con soluzioni per tutte le istanze più piccole del problema, fermandosi non appena si raggiunge il valore a cui si è interessati. A questo punto si legge il valore e lo restituisce. Fondamentalmente, ogni sotto-problema è sempre e solo una volta risolto.

Si noti che l'approccio adottato in virahanka2 () è quello di risolvere i problemi più piccoli sulla strada giusta per risolvere i problemi più grandi. Di conseguenza, questo è noto come approccio ascendente alla programmazione dinamica. Purtroppo risulta essere piuttosto dispendioso per alcune applicazioni, poiché può calcolare soluzioni ai sotto-problemi che non sono necessari per risolvere il problema principale. Questo calcolo sprecato può essere evitato utilizzando l'approccio top-down della programmazione dinamica, che è illustrata nella funzione virahanka3 () in 4.12. A differenza dell' approccio ascendente, questo approccio è ricorsivo. Evita lo spreco enorme di virahanka1 () per verificare se è stato precedentemente memorizzato il risultato. Se no, calcola il risultato ricorsivo e lo memorizza nella tabella. L'ultimo passo è quello di restituire il risultato memorizzato. L'ultimo metodo, in virahanka4 (), è quello di utilizzare un Python "decoratore" chiamato memoize, che si occupa del lavoro di pulizia svolto da virahanka3 () senza ingombrare il programma. Questo processo "memoization" memorizza il risultato di ogni chiamata precedente alla funzione con i parametri che sono stati utilizzati. Se la funzione viene successivamente chiamata con gli stessi parametri, restituisce il risultato memorizzato invece di ricalcolarlo . (Questo aspetto della sintassi Python va oltre lo scopo di questo libro.)

Questo conclude la nostra breve introduzione alla programmazione dinamica. Noi lo incontriamo di nuovo in 8.4.

4.8 Un Esempio di librerie Python

Python dispone di centinaia di librerie esterne, pacchetti software specializzati che estendono le funzionalità di Python. NLTK è una libreria del genere. Per realizzare la potenza di programmazione Python, è necessario acquisire familiarità con diverse altre librerie. La maggior parte di questi dovrà essere installato manualmente sul computer.

Matplotlib

Python ha alcune librerie che sono utili per la visualizzazione dei dati linguistici. Il pacchetto supporta matplotlib sofisticate funzioni di tracciatura con un interfaccia in stile MATLAB, ed è disponibile da http://matplotlib.sourceforge.net/.

Fino ad ora ci siamo concentrati su presentazione testuale e l'uso di istruzioni di stampa formattati per ottenere l'output allineato in colonne. E 'spesso molto utile per visualizzare i dati numerici in forma grafica, in quanto ciò rende spesso più facile individuare i modelli. Ad esempio, al punto 3.7 abbiamo visto una tabella di numeri che mostrano la frequenza di particolari verbi modali nel Brown Corpus, classificati per genere. Il programma in 4.13 presenta le stesse informazioni in formato grafico. L'uscita è mostrata in 4.14 (figura colore nel display grafico).

```
colors = 'rgbcmyk' # red, green, blue, cyan, magenta, yellow, black

def bar_chart(categories, words, counts):
    "Plot a bar chart showing counts for each word by category"
    import pylab
    ind = pylab.arange(len(words))
    width = 1 / (len(categories) + 1)
    bar_groups = []
    for c in range(len(categories)):
        bars = pylab.bar(ind+c*width, counts[categories[c]], width,
                         color=colors[c % len(colors)])
        bar_groups.append(bars)
    pylab.xticks(ind+width, words)
    pylab.legend([b[0] for b in bar_groups], categories, loc='upper left')
```

```
    pylab.ylabel('Frequency')
    pylab.title('Frequency of Six Modal Verbs by Genre')
    pylab.show()
```

```
 >>> genres = ['news', 'religion', 'hobbies', 'government', 'adventure']
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> cfdist = nltk.ConditionalFreqDist(
...             (genre, word)
...             for genre in genres
...             for word in nltk.corpus.brown.words(categories=genre)
...             if word in modals)
...
>>> counts = {}
>>> for genre in genres:
...     counts[genre] = [cfdist[genre][word] for word in modals]
>>> bar_chart(genres, modals, counts)
```

Esempio 4.13 (code_modal_plot.py): Figura 4.13: Frequenza dei verbi modali in diverse sezioni del Brown Corpus

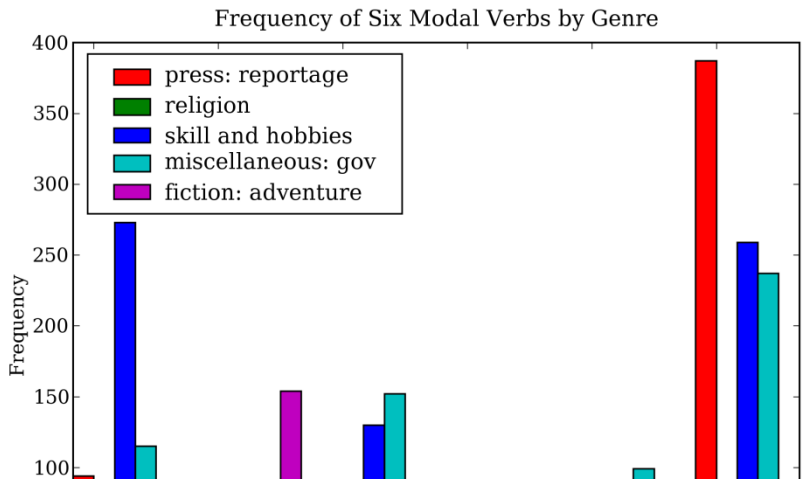


Figura 4.14: Istogramma indica la frequenza di verbi modali nelle varie sezioni del Brown Corpus: questa visualizzazione è stata prodotta dal programma in 4.13.

Dall' istogramma è immediatamente evidente che può e deve avere le frequenze relative quasi identiche. Lo stesso vale per potuto e potrebbe.

È anche possibile generare tali visualizzazioni di dati al volo. Ad esempio, una pagina web con modulo di input potrebbe consentire ai visitatori di specificare i parametri di ricerca, inviare il

modulo, e vedere una visualizzazione generata dinamicamente. Per fare questo dobbiamo specificare il backend per Agg matplotlib, che è una libreria per la produzione di raster (pixel) immagini. Poi si usano tutte le modalità medesime PyLab di prima, ma invece di visualizzare il risultato su un terminale grafico con pylab.show (), lo salviamo in un file utilizzando pylab.savefig (). Specifichiamo il nome del file e dpi, quindi stampare il codice HTML che indirizza il browser Web per caricare il file.

```
 >>> import matplotlib

>>> matplotlib.use('Agg')

>>> pylab.savefig('modals.png')

>>> print 'Content-Type: text/html'

>>> print

>>> print '<html><body>'

>>> print '<img src="modals.png"/>'

>>> print '</body></html>'
```

NetworkX

Il pacchetto NetworkX è per definire e manipolare strutture consistenti di nodi e spigoli, noti come grafici. E 'disponibile da https://networkx.lanl.gov/. NetworkX può essere utilizzato in combinazione con matplotlib di visualizzare le reti, quali WordNet (la rete semantica abbiamo introdotto in 2.5). Il programma in 4.15 inizializza un grafico vuoto poi attraversa la gerarchia WordNet iperonimo aggiungendo bordi al grafico. Si noti che l'attraversamento è ricorsivo, applicando la tecnica di programmazione discussa in 4.7. La visualizzazione risultante è mostrata in 4.16.

```
 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)


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


def graph_draw(graph):
    nx.draw_graphviz(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()



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

Esempio 4.15 (code_networkx.py): Figura 4.15: Utilizzo del NetworkX e Biblioteche matplotlib

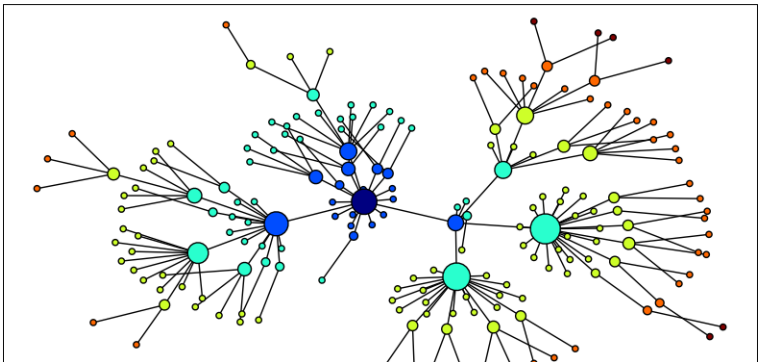


Figura 4.16: Visualizzazione con NetworkX e matplotlib: Parte della gerarchia WordNet hypernym viene visualizzata, a partire dog.n.01 (il più scuro nodo al centro); le dimensioni dei nodi si basano sul numero di figli del nodo, e il colore è sulla base della distanza del nodo da dog.n.01; questa visualizzazione è stata prodotta dal programma in 4.15.

csv

L'analisi del linguaggio implica spesso tabulati di dati, contenenti informazioni sugli elementi lessicali, o dai partecipanti in uno studio empirico, o le caratteristiche linguistiche estratte `< da un corpus. Ecco un frammento di un lessico semplice, in formato CSV:

sleep, sli:p, v.i, a condition of body and mind ...

walk, wo:k, v.intr, progress by lifting and setting down each foot ...

wake, weik, intrans, cease to sleep

Possiamo usare la biblioteca CSV di Python per leggere e scrivere i file memorizzati in questo formato. Per esempio, siamo in grado di aprire un file CSV denominato lexicon.csv 1 e scorrere le righe 2:

```
>>> import csv
>>> input_file = open("lexicon.csv", "rb") 1
>>> for row in csv.reader(input_file): 2
...     print row
['sleep', 'sli:p', 'v.i', 'a condition of body and mind ...']
['walk', 'wo:k', 'v.intr', 'progress by lifting and setting down each foot ...']
['wake', 'weik', 'intrans', 'cease to sleep']
```

Ogni riga è semplicemente una lista di stringhe. Se i campi contengono dati numerici, questi verranno visualizzati come stringhe, e dovranno essere convertiti con int () o float ().

NumPy

Il pacchetto NumPy contribuisce in maniera sostanziale alll'elaborazione numerica in Python. NumPy è un oggetto matrice multi-dimensionale, che è facile da inizializzare e accedere a:

```
>>> from numpy import array
>>> cube = array([ [[0,0,0], [1,1,1], [2,2,2]],
...             [[3,3,3], [4,4,4], [5,5,5]],
...             [[6,6,6], [7,7,7], [8,8,8]] ])
>>> cube[1,1,1]
4
>>> cube[2].transpose()
array([[6, 7, 8],
       [6, 7, 8],
       [6, 7, 8]])
```

```
>>> cube[2,1:]
array([[7, 7, 7],
       [8, 8, 8]])
```

NumPy include funzioni di algebra lineare. Ecco effettuare decomposizione in valori singolari di una matrice, operazione utilizzata in analisi semantica latente per aiutare a identificare i concetti impliciti in una collezione di documenti.

```
 >>> from numpy import linalg
>>> a=array([[4,0], [3,-5]])
>>> u,s,vt = linalg.svd(a)
>>> u
array([[-0.4472136 , -0.89442719],
       [-0.89442719,  0.4472136 ]])
>>> s
array([ 6.32455532,  3.16227766])
>>> vt
array([[-0.70710678,  0.70710678],
       [-0.70710678, -0.70710678]])
```

Il Pacchetto NLTK di nltk.cluster di raggruppamento fa ampio uso di matrici NumPy, e include il supporto per k-means, Gaussian EM clustering, gruppo raggruppamento medio agglomerativo e dendrogramma trame. Per ulteriori informazioni, digitare help (nltk.cluster).

Altre librerie Python

Ci sono molte altre librerie Python, ed è possibile effettuare ricerche per loro con l'aiuto del Python Package Index http://pypi.python.org/. Molte biblioteche forniscono un'interfaccia al software esterno, come ad esempio database relazionali (ad esempio mysql-python) e collezioni di documenti di grandi dimensioni (ad esempio, PyLucene). Molte altre biblioteche danno accesso a formati di file come PDF, MSWord, e XML (pypdf, pywin32, xml.etree), i feed RSS (ad esempio feedparser), e la posta elettronica (ad esempio imaplib, e-mail).

4.9 Sommario

- L'assegnazione di Python e il parametro di passaggio usano riferimenti agli oggetti di uso: ad esempio se a è un elenco e assegna b = a, quindi qualsiasi operazione modificherà B, e viceversa.
- Il test di funzionamento is se due oggetti sono identici agli oggetti interni, mentre == verifica se due oggetti sono equivalenti. Questa distinzione è parallela al tipo-token distinzione.
- Stringhe, liste e tuple sono diversi tipi di oggetto sequenza, a sostegno delle operazioni comuni come l'indicizzazione, affettatrici, len () , sorted (), e le prove che utilizzano l'appartenenza in.
- Siamo in grado di scrivere il testo in un file, aprire il file per la scrittura oFile = open ('output.txt', 'w'), quindi l'aggiunta di contenuti al ofile.write file ("Monty Python"), e, infine, la chiusura del file di oFile . close ().
- Uno stile di programmazione dichiarativa di solito produce maggiore compattezza, codice leggibile; variabili di ciclo manualmente incrementati di solito sono inutili, quando le sequenze devono essere elencate, utilizzate enumerate ().
- Le funzioni sono un'astrazione di programmazione essenziale: i concetti chiave per comprendere sono il parametro di passaggio, applicazione variabile, e docstring.
- Una funzione serve come dominio: i nomi definiti all'interno di una funzione non sono visibili fuori di tale funzione, a meno che tali nomi vengono dichiarati globali.
- I moduli consentono materiali logicamente correlati a essere localizzati in un file. Un modulo serve come un dominio: i nomi definiti in un modulo - come variabili e funzioni - non sono visibili ad altri moduli, a meno che tali nomi vengono importati.
- La programmazione dinamica è una tecnica di progettazione algoritmo utilizzata ampiamente in NLP che memorizza i risultati dei calcoli precedenti, al fine di evitare inutile ricalcolo.

4.10 Approfondimenti

Questo capitolo ha toccato molti argomenti in programmazione, alcuni specifici per Python, e alcuni molto generali. Abbiamo appena scalfito la superficie, e si consiglia di leggere di più su questi argomenti, a partire dai materiali ulteriori per questo capitolo disponibili presso http://www.nltk.org/.

Il sito web Python fornisce un'ampia documentazione. È importante comprendere le funzioni incorporate e i tipi standard, descritti in http://docs.python.org/library/functions.html e http://docs.python.org/library/stdtypes.html. Abbiamo imparato a conoscere i generatori e la loro importanza per l'efficienza, per informazioni su iteratori, un argomento strettamente

correlato, vedere http://docs.python.org/library/itertools.html. Consultate il vostro libro preferito di Python per ulteriori informazioni su questi argomenti. Una risorsa eccellente per l'utilizzo di Python per l'elaborazione multimediale, tra cui il lavoro con i file audio, è (Guzdial, 2005).

Quando si utilizza la documentazione in linea di Python, bisogna essere consapevoli del fatto che la versione installata potrebbe essere diversa dalla versione della documentazione che si sta leggendo. Si può facilmente verificare quale versione si ha, con import sys; sys.version. Versione documentazione specifica è disponibile presso http://www.python.org/doc/versions/.

La progettazione di algoritmi è un campo ricco all'interno della scienza del computer. Alcuni buoni punti di partenza sono (Harel, 2004), (Levitin, 2004), (Knuth, 2006). Indicazioni utili sulla pratica dello sviluppo del software sono fornite in (Hunt & Thomas, 2000) e (McConnell, 2004).

4.11 Esercizi

1. ☼ Per saperne di più sugli oggetti di sequenza utilizzando funzione di aiuto di Python. In aiuto dell'interprete, il tipo (str), help (lista), e contribuire (tupla). Questo vi darà un elenco completo delle funzioni supportate da ciascun tipo. Alcune funzioni hanno nomi speciali fiancheggiate da sottolineare, come la documentazione di aiuto mostra, ogni tale funzione corrisponde a qualcosa di più familiare. Per esempio x.__getitem__ (y) è solo un prolisso modo di dire x [y].
2. ☼ Identificare tre operazioni che possono essere eseguite su entrambe le tuple e liste. Identificate tre operazioni lista che non possono essere eseguite su tuple. Nome di un contesto in cui l'utilizzo di un elenco, invece di una tupla genera un errore di Python.
3. ☼ Scopri come creare una tupla composta da un singolo elemento. Ci sono almeno due modi per farlo.
4. ☼ Crea parole lista = ['is', 'NLP', 'fun', '?']. Utilizzare una serie di istruzioni di assegnazione (words [1] = words [2]) e una variabile temporanea tmp per trasformare questa lista nella lista ['NLP', 'is', 'fun', '!']. Ora fate la stessa trasformazione utilizzando l'assegnazione tupla.
5. ☼ Leggi la costruzione della funzione di confronto cmp, con l'aiuto digitando (cmp). Come si differenzia nel comportamento da parte degli operatori di confronto?
6. ☼ Il metodo utilizzato per creare una finestra scorrevole di n-grammi si comportarsi correttamente per i due casi limite: n = 1 e n = len (sent)?
7. ☼ Abbiamo osservato che quando le stringhe vuote e gli elenchi vuoti si verificano nella parte condizione di una clausola if, si restituiscono false. In questo caso, si dice che si verificano in un contesto Boolean. Sperimentare con diversi tipi di espressioni non booleane in contesti Boolean, e vedere se la valutazione è True o False.
8. ☼ Utilizzare gli operatori di disuguaglianza per confrontare le stringhe, ad esempio, <'Monty' 'Python'. Che cosa succede quando si fa 'Z' <'a'? Provate coppie di stringhe che hanno un prefisso comune, ad esempio 'Monty' <'Montague'. Leggi su "lessicografico liste", al fine di capire che cosa sta succedendo qui. Provate a confrontare oggetti strutturati, es ('Monty', 1) <('Monty', 2). Questo si comporta come previsto?
9. ☼ Scrivi il codice che rimuove gli spazi bianchi all'inizio e alla fine di una stringa, e normalizza gli spazi bianchi tra le parole per essere un singolo carattere di spazio.

1. eseguire questa operazione utilizzando split () e join ()
2. eseguire questa operazione utilizzando le sostituzioni delle espressioni Regolari.

10. ☼ Scrivere un programma per ordinare le parole per lunghezza. Definire una cmp_len funzione di supporto che utilizza la funzione di confronto cmp su lunghezze di parola.
11. ◑ Creare un elenco di parole e memorizzalo in una variabile sent1. Ora assegnare sent2 = sent1 . Modificare una delle voci in sent1 e verificare che sent2 è cambiato.

    1. Ora provate lo stesso esercizio, ma invece di assegnare sent2 = sent1 [:]. Modifica sent1 di nuovo e vedere cosa succede a sent2. Spiega.
    2. Ora definire text1 ad essere una lista di liste di stringhe (ad esempio, per rappresentare un testo composto da più frasi. Ora assegnate text2 = text1. [:], Assegnare un nuovo valore a una delle parole, ad esempio text1 [1] [1] = 'Monty'. Controllare che cosa ha fatto a questo text2. Spiega.
    3. Caricare deepcopy Python () la funzione (cioè da importazione deepcopy copia), consultare la relativa documentazione, e prova che si fa una nuova copia di un qualsiasi oggetto.
12. ◑ Inizializzare un n-by-m lista di liste di stringhe vuote utilizzando la moltiplicazione lista, per es word_table = [['] * n] * m. Che cosa succede quando si imposta uno dei suoi valori, ad esempio word_table [1] [2] = "hello"? Spiegate perché questo accade. Ora scrivere un'espressione che utilizza range () per costruire una lista di liste, e dimostrare che non presenta questo problema.
13. ◑ Scrivere il codice per inizializzare un array bidimensionale di set denominati word_vowels ed elaborare un elenco di parole, aggiungendo ogni parola per word_vowels [l] [v] dove l è la lunghezza della parola e v è il numero di vocali che contiene.
14. ◑ Scrivere una funzione novel10 (text) che consente di stampare qualsiasi parola che è apparsa nell'ultimo 10% di un testo che non era stato rilevato in precedenza.
15. ◑ Scrivere un programma che prende una frase espressa come una singola stringa, dividi e conta le parole. Scarica per stampare ogni parola e la frequenza della parola, uno per riga, in ordine alfabetico.
16. ◑ Leggi su Gematria, un metodo per assegnare i numeri in parole, e per la mappatura tra le parole che hanno lo stesso numero per scoprire il significato nascosto dei testi (http://en.wikipedia.org/wiki/Gematria, http://essenes .net / gemcal.htm).

    1. Scrivi una funzione gematria () che somma i valori numerici delle lettere di una parola, in base ai valori delle lettere in letter_vals:

```
>>> letter_vals = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':80, 'g':3, 'h':8,

... 'i':10, 'j':10, 'k':20, 'l':30, 'm':40, 'n':50, 'o':70, 'p':80, 'q':100,

... 'r':200, 's':300, 't':400, 'u':6, 'v':6, 'w':800, 'x':60, 'y':10, 'z':7}
```

    2. Elabora un corpus (nltk.corpus.state_union ad esempio) e per ogni documento, contare quante parole hanno il numero 666.

    3. Scrivere una funzione di decodifica () per elaborare un testo, in modo casuale sostituendo le parole con i loro equivalenti Gematria, al fine di scoprire il "significato nascosto" del testo.

17. ◑ Scrivere una funzione abbreviata (text, n) per elaborare un testo, omettendo la n più frequente parole del testo. Come è leggibile?
18. ◑ Scrivere il codice per stampare un indice per un lessico, che permette a qualcuno di cercare le parole in base al loro significato (o pronunce; qualunque proprietà sono contenute nelle voci lessicali).
19. ◑ Scrivi una lista di comprensione che ordina una lista di synset WordNet per la vicinanza ad un synset dato. Per esempio, dati i synset minke_whale.n.01, orca.n.01, novel.n.01 e tortoise.n.01, ordinarli secondo la loro shortest_path_distance () da right_whale.n.01.
20. ◑ Scrivere una funzione che prende una lista di parole (contenente duplicati) e restituisce una lista di parole (senza duplicati) in ordine decrescente di frequenza. Ad esempio se la lista di input conteneva 10 istanze della parola tavolo e 9 istanze della parola sedia, quindi tavolo sembrerebbe prima di sedia in lista di output.
21. ◑ Scrivere una funzione che prende un testo e un vocabolario come i suoi argomenti e restituisce l'insieme di parole che appaiono nel testo, ma non nel vocabolario. Entrambi gli argomenti possono essere rappresentati come liste di stringhe. Si può fare questo in una sola riga, utilizzando set.difference ()?
22. ◑ Importare l' itemgetter () funzione dal modulo operatore in libreria standard di Python (cioè da operatore importa itemgetter). Creare un elenco di parole che contengono più parole. Ora provate a chiamare: sorted (words, key= itemgetter (1)), e sorted (words, key = itemgetter (-1)). Spiega cosa itemgetter () sta facendo.
23. ◑ Scrivi una ricerca di funzione ricorsiva (trie, key) che cerca key in trie, e restituisce il valore che trova. Estendere la funzione per restituire una parola quando è univocamente determinata dal suo prefisso (ad esempio vanguard è l'unica parola che inizia con vang-, in modo di ricerca (trie, 'vang') dovrebbe restituire la stessa cosa di ricerca (trie, 'vanguard')).
24. ◑ Leggi su "collegamento parola chiave" (capitolo 5 (Scott & Tribble, 2006)). Estrai parole chiave da NLTK Shakespeare Corpus e utilizzando il pacchetto NetworkX, trama delle reti di linkage parole chiave.
25. ◑ Leggi della distanza stringa di modifica e l'algoritmo di Levenshtein. Prova l'implementazione fornita in nltk.edit_dist (). In che modo questo utilizza la programmazione dinamica? Utilizza l'approccio bottom-up o top-down? [Vedi anche http://norvig.com/spell-correct.html].
26. ◑ I numeri catalani sorgono in molte applicazioni di matematica combinatoria, incluso il conteggio di alberi di analisi (8.6). La serie può essere definita come segue: C0 = 1, e Cn +1 = Σ0 .. n (cicn-i).
    1.Scrivere una funzione ricorsiva per calcolare nth numero catalano Cn.
    2.Ora scrivere un'altra funzione che fa questo calcolo utilizzando la programmazione dinamica.
    3.Utilizzare il modulo timeit per confrontare le prestazioni di queste funzioni al crescere di n.

27. ★ riprodurre alcuni dei risultati di (Zhao e Zobel, 2007) in materia di identificazione dell'autore.
28. ★ Studia la scelta lessicale del genere specifico, e vedi se è in grado di riprodurre alcuni dei risultati di http://www.clintoneast.com/articles/words.php
29. ★ Scrivere una funzione ricorsiva che stampa piuttosto un trie in ordine alfabetico, ad esempio:

        chair: 'flesh'

        ---t: 'cat'

        --ic: 'stylish'

    ---en: 'dog'.

30. ★ Con l'aiuto della struttura dati trie, scrivere una funzione ricorsiva che elabora il testo, individuando il punto di unicità in ogni parola, e scartando il resto di ogni parola. Quanta compressione vuol dare? Quanto è leggibile il testo risultante?
31. ★ Ottenere un testo crudo, sotto forma di un unico, lungo filo. Utilizzare il modulo textwrap Python per suddividerlo in più righe. Ora scrivere il codice per aggiungere spazi aggiuntivi tra le parole, per giustificare l'uscita. Ogni riga deve avere la stessa larghezza, e gli spazi devono essere approssimativamente uniformemente distribuiti tra le linee. Nessuna linea può iniziare o terminare con uno spazio.
32. ★ Sviluppare uno strumento di semplice riepilogo di estrazione, che consente di stampare le frasi di un documento che contengono la più alta frequenza totale di parola. Utilizzare FreqDist () per contare le frequenze di parole, e utilizzare sum per riassumere le frequenze delle parole in ogni frase. Classificare le frasi in base al loro punteggio. Infine, stampare l' n punteggio più alto nelle frasi nell'ordine dei documenti. Esaminare attentamente la struttura del programma, in particolare il suo approccio a questo ordinamento doppio. Assicurarsi che il programma è scritto nel modo più chiaro possibile.
33. ★ Leggi l'articolo seguente sull' orientamento semantica degli aggettivi. Utilizzare il pacchetto NetworkX per visualizzare una rete di aggettivi con i bordi per indicare lo stesso vs diverso orientamento semantico. http://www.aclweb.org/anthology/P97-1023.
34. ★ Progettare un algoritmo per trovare le "frasi statisticamente improbabili" di una collezione di documenti. http://www.amazon.com/gp/search-inside/sipshelp.html
35. ★ Scrivere un programma per implementare un algoritmo di forza bruta per scoprire caselle di parole, una specie di n × n: cruciverba matematica in cui la voce nella riga nth è la stessa voce nella colonna n. Per la discussione, vedere http://itre.cis.upenn.edu/ ~ myl/languagelog/archives/002679.html

Circa questo documento ...

Questo documento è stato realizzato lunedi 15 ott 2012 16:46:09 EST

Capitolo 5

Capitolo 6 Imparare a classificare un testo

(traduzione iniziale Roberta Toscano)

Rilevare i modelli è una parte centrale del Natural Language Processing. Parole che terminano in –ed tendono a essere verbi passati (capitoli cinque). Il frequente uso di will è indicativo del nuovo testo (capitoli tre). Questi osservabili modelli, struttura e frequenza delle parole, sono in rapporto con particolari aspetti di significato, come il tempo e l'argomento. Come facciamo a sapere, dove dobbiamo cominciare a guardare, quali aspetti della forma associare a quali aspetti del significato?

L'obiettivo di questo capitolo è domandarsi le seguenti questioni:

1. Come possiamo individuare particolari caratteristiche di dati linguistici che sono salienti per classificarlo?
2. Come possiamo automaticamente costruire modelli di linguaggio che possono essere utilizzati per eseguire attività di elaborazione del linguaggio?
3. Che cosa possiamo imparare a proposito del linguaggio da questi modelli?

Lungo il percorso avremo modo di studiare alcune importanti tecniche di apprendimento, tra cui gli alberi di decisione, classificatori Naive Bayes, e classificatori di massima entropia. Noi vorremo sorvolare le basi matematiche e statistiche di queste tecniche, focalizzandoci invece su come e quando usarle. Prima di guardare a questi metodi, abbiamo bisogno di renderci conto dell'ampia portata per quest'argomento.

6.1 Classificazione Supervisionata

La classificazione è il compito di scegliere la corretta etichetta di classe per un dato input. Nel compito di classificazione di base, ogni imput è considerato separatamente da tutti gli altri e l'insieme di etichette è definito in anticipo. Alcuni esempi di attività di classificazione sono:

- Decidere se un messaggio è spam o meno.
- Decidere quale sia l'argomento di un articolo giornalistico da un elenco fisso di aree tematiche, come "sport", "tecnologia" e "politica"
- Decidere se un dato evento della banca delle parole è stato usato per riferirsi a una riva del fiume, un istituto finanziario, all'atto di ribaltamento da una parte, o all'atto di depositare qualcosa in un istituto finanziario.

Nel compito di classificazione di base c'è un certo numero di interessanti varianti. Per esempio, nella classificazione multi-classe, ogni proposta può essere assegnata a più etichette; nella classificazione open-classe, l'insieme di etichette non è definita in anticipo, e nella classificazione in sequenza, un elenco d'input sono classificati congiuntamente.

Un classificatore è **supervisionato** se è costruito basandosi sul corpora di lavoro, contenente l'etichetta giusta per ogni input. Lo schema utilizzato dalla classificazione supervisionata è mostrato in Figura 6.1 .

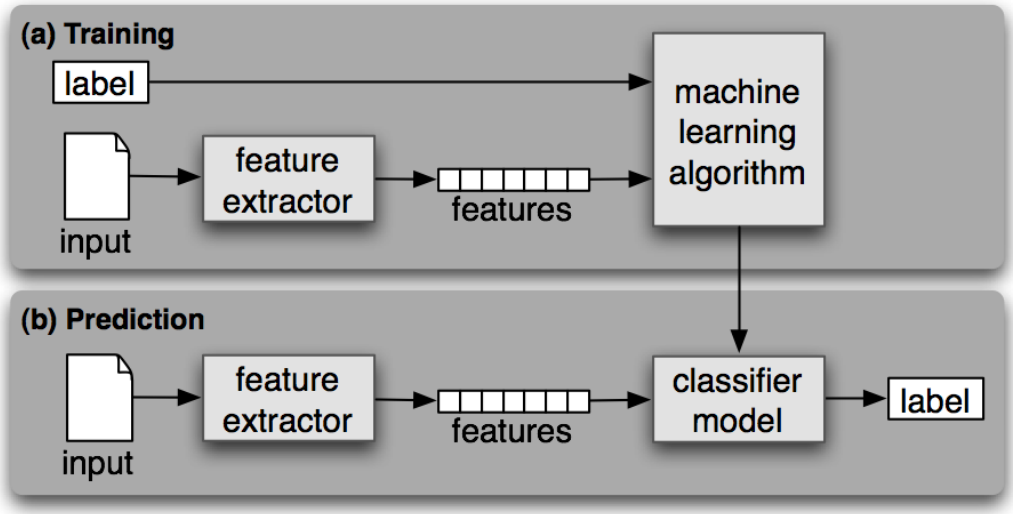


Figura 6.1 : classificazione supervisionata. (a) Durante la pratica, un estrattore di funzione è utilizzato per convertire ogni valore d'input a un set di funzione. Questi set di funzione, che catturano le informazioni di base su ogni input, sarebbero stati usati per classificarlo, ciò sarà discusso nella prossima sezione. Coppie di set di funzioni e di etichette sono alimentate nella macchina di apprendimento dell'algoritmo per generare un modello. (b) Durante la predizione, lo stesso estrattore di funzione è utilizzato per convertire input invisibili ai set di funzione. Questi set di funzione sono poi immessi nel modello, che genera le etichette previste.

Nel resto di questa sezione, vedremo come i classificatori possono essere impiegati per risolvere una vasta gamma di compiti. La nostra discussione non vuole essere comprensiva, ma intende dare un campione rappresentativo di operazioni che possono essere eseguite con l'aiuto di classificatori di testo.

Identificazione di genere

Nella Sezione 2.4 abbiamo visto che i nomi maschili e femminili hanno alcune caratteristiche distintive. I nomi che terminano in, *a* , *e*, *i,* è probabile che siano di sesso femminile, mentre i nomi che terminano in *k* , *o* , *r* , *s e t* sono probabili che siano maschili. Costruiamo più precisamente un classificatore per modellare queste differenze.

Il primo passo nella creazione di un classificatore è decidere quali **caratteristiche** d'imput sono rilevanti, e come **codificarle**. Per esempio, inizieremo guardando la lettera finale di un determinato nome. Il seguente estrattore di funzione costruisce un dizionario contenente le informazioni pertinenti di un determinato nome:

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

Il dizionario, conosciuto come un **set di funzione** ,passa dalle caratteristiche dei nomi ai loro valori. Le caratteristiche dei nomi sono stringhe che in genere forniscono una breve descrizione leggibile della funzione. Le caratteristiche dei valori hanno semplici qualità, come booleani, numeri e stringhe.

Nota

La maggior parte dei metodi di classificazione richiede che le funzioni devono essere codificate usando le qualità semplici di valore, come booleani, numeri e stringhe. Si noti che, solo perché una caratteristica ha una qualità semplice, non significa necessariamente che il valore della funzione è semplice da esprimere o calcolare, anzi, è anche possibile utilizzare i valori molto complessi, come la produzione di un secondo classificatore supervisionato.

Ora che abbiamo definito un estrattore di funzione, abbiamo bisogno di preparare una lista di esempi e le corrispondenti etichette di classe:

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

In seguito, utilizzare l'estrattore di funzione per elaborare i dati dei nomi, e dividere il risultato della lista del set di funzione in un **training set** e in un **test set** .Il training set è utilizzato per formare un nuovo classificatore "Naive Bayes".

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

Impareremo di più sul classificatore Naive Bayes nel capitolo più avanti. Per ora, limitiamoci a provarlo su alcuni nomi che non compaiono nei suoi dati di lavoro:

```
>>> classifier.classify(gender_features('Neo'))
'male'
>>> classifier.classify(gender_features('Trinity'))
'female'
```

Si noti che i nomi dei personaggi di *The Matrix* sono correttamente classificati. Anche se questo film di fantascienza è incominciato nel 1992, è ancora conforme alle nostre attese sui nomi e generi. Siamo in grado di valutare sistematicamente il classificatore su una quantità molto maggiore di dati invisibili:

```
>>> print nltk.classify.accuracy(classifier, test_set)
0.758
```

Infine, possiamo esaminare il classificatore per determinare quali caratteristiche più efficaci ha trovato per distinguere il genere dei nomi:

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
             last_letter = 'a'            female : male   =     38.3 :
1.0
             last_letter = 'k'              male : female =     31.4 :
1.0
```

```
              last_letter = 'f'              male : female =     15.3 :
1.0
              last_letter = 'p'              male : female =     10.6 :
1.0
              last_letter = 'w'              male : female =     10.6 :
1.0
```

Questo elenco mostra che i nomi del training set che terminano in "a" sono femminili 38 volte più spesso di quelli maschili, ma i nomi che terminano in "k" sono maschili 31 volte più spesso di quelli femminili. Questi rapporti sono noti come **rapporti di probabilità** , e può essere utile per confrontare diversi esiti di rapporti funzionali.

Nota

Il vostro turno: Modificare le caratteristiche del *gender_features ()* fornire al classificatore le caratteristiche che codificano la lunghezza del nome, la sua prima lettera, e altre caratteristiche. Riqualificare il classificatore con queste nuove caratteristiche, e testare la sua accuratezza.

Quando si lavora su corpi di grandi dimensioni, la costruzione di un unico elenco che contiene le caratteristiche di ogni caso è usata per una grande quantità di memoria. In questi casi, utilizzare la funzione *nltk.classify.apply_features* , che restituisce un oggetto che si comporta come un elenco, ma non memorizza tutta la funzione impostate nella memoria.

```
>>> from nltk.classify import apply_features
>>> train_set = apply_features(gender_features, names[500:])
>>> test_set = apply_features(gender_features, names[:500])
```

Scegliere le giuste caratteristiche

Selezionando delle rilevanti funzioni e decidendo come codificarle per un metodo di apprendimento, può avere un enorme impatto sulla capacità di apprendimento per estrarre un buon modello. Gran parte dell'interessante lavoro nella costruzione di un classificatore è decidere quali funzioni potrebbero essere rilevanti, e in che modo possiamo rappresentarle. Anche se è spesso possibile ottenere prestazioni decenti utilizzando un insieme abbastanza semplice ed evidente di caratteristiche, ci sono in genere vantaggi efficaci per essere usato attentamente costruendo figure per una conoscenza approfondita del compito a portata di mano.

In genere, estrattori di funzioni sono costruiti attraverso un processo di tentativi ed errori, guidato da intuizioni su quello che è pertinente al problema. È comune iniziare con un approccio al "lavandino" , che include tutte le caratteristiche cui possiamo pensare, e poi controllare per vedere quali caratteristiche sono in realtà utili. Chiamiamo questo approccio con il nome *gender features* nell' Esempio 6.2 .

```
def gender_features2(name):
    features = {}
    features["firstletter"] = name[0].lower()
    features["lastletter"] = name[-1].lower()
    for letter in 'abcdefghijklmnopqrstuvwxyz':
        features["count(%s)" % letter] = name.lower().count(letter)
        features["has(%s)" % letter] = (letter in name.lower())
```

```
    return features
>>> gender_features2('John')
{'count(j)': 1, 'has(d)': False, 'count(b)': 0, ...}
```

Tuttavia, di solito ci sono limiti al numero di caratteristiche che si dovrebbero utilizzare con l'apprendimento di un dato algoritmo, se si forniscono troppe caratteristiche, l'algoritmo avrà una maggiore possibilità di far valere idiosincrasie dei vostri dati di lavoro. Questo problema è noto come **overfitting** , e può essere particolarmente difficile quando si lavora con piccoli gruppi di formazione. Ad esempio, se lavoriamo su un classificatore Naive Bayes utilizzando l'estrattore di funzione mostrato nell' Esempio 6.2 , il set di lavoro dell' Overfit è relativamente piccolo, risulterebbe in un sistema la cui accuratezza è circa 1% inferiore alla precisione di un classificatore, che mostra solo attenzione all'ultima lettera di ogni nome:

```
>>> featuresets = [(gender_features2(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.748
```

Dopo una prima serie di caratteristiche è stato scelto un metodo molto produttivo per affinare il set di funzionalità, è **l'analisi degli errori** . Per prima cosa, si seleziona un **development set** , contenente il corpo di dati per creare il modello. Questo set di sviluppo è quindi suddiviso in **training set** e il **dev-test set**.

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

Il set di lavoro è utilizzato per il lavoro del modello, e il dev-test set è utilizzato per eseguire l'analisi degli errori. Il test set serve nella nostra valutazione finale del sistema. Per motivi discussi di seguito, è importante che ci avvaliamo di un separato dev-test set per l'analisi degli errori, piuttosto che utilizzare il test set. La divisione del corpo dei dati in diversi sottoinsiemi è mostrato in Figura 6.3 .

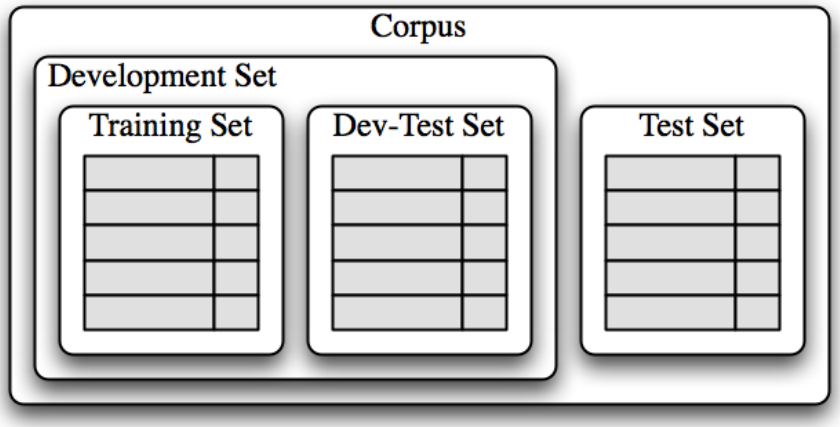


Figura 6.3 : Organizzazione del corpo dei dati per classificatori di formazione sotto la supervisione. Il corpo di dati è diviso in due gruppi: il set di sviluppo, e il test set. Il set di sviluppo è spesso ulteriormente suddiviso in un set di lavoro e un deve-test set. Dopo aver

diviso il corpo in set di dati appropriati, formiamo un modello con il training set e quindi lo eseguiamo sul dev test set:

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]

>>> classifier = nltk.NaiveBayesClassifier.train(train_set)

>>> print nltk.classify.accuracy(classifier, devtest_set)
0.765
```

Utilizzando il dev-test set, siamo in grado di generare un elenco degli errori che il classificatore fa quando afferma il genere del nome:

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

Possiamo quindi esaminare i singoli casi di errore in cui il modello asserisce l'etichetta sbagliata, e cercare di determinare quali informazioni supplementari le consentirebbe di prendere la decisione giusta. La classificazione dei nomi che abbiamo costruito genera circa 100 errori sul corpo del dev-test:

```
>>> for (tag, guess, name) in sorted(errors):
...     print 'correct=%-8s guess=%-8s name=%-30s' % (tag, guess, name)
  ...
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
  ...
```

Guardando attraverso questa lista di errori rende chiaro che alcuni suffissi sono più di una lettera, il che può essere indicativo per il sesso dei nomi. Ad esempio, i nomi che terminano in *yn* sembrano essere prevalentemente femminili, nonostante che, i nomi che terminano in *n* tendono a essere di sesso maschile, e nomi che terminano in *ch* sono di solito di sesso maschile, anche se i nomi che terminano in *h* tendono a essere femminili. Dobbiamo quindi regolare il nostro estrattore di funzione per includere funzioni con due lettere di suffisso:

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

Ricostruendo il classificatore con il nuovo estrattore di funzione, si vede che le prestazioni dei dati del dev-test migliorano di quasi 3 punti percentuali (dal 76,5% al 78,2%):

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

Questa procedura di analisi di errore può essere ripetuta, controllando i modelli negli errori che sono fatti dal classificatore recentemente migliorato. Ogni volta che la procedura di analisi di errore si ripete, si dovrebbe selezionare una scissione diversa dev-test/training, per garantire che il classificatore non inizia a riflettere idiosincrasie sul set del dev-test.

Una volta che noi abbiamo usato il dev-test per aiutarci a sviluppare il modello, non si può più credere che ci darà un'idea precisa di come il modello si comporta su nuovi dati. È quindi importante mantenere l'insieme di test separato, e non utilizzarlo fino a quando il nostro modello di sviluppo non sia completo. A quel punto, si può utilizzare il test impostato a valutare bene se il nostro modello si esibirà su nuovi valori d'input.

Documento di Classificazione

Nella Sezione 2.1 , abbiamo visto diversi esempi di corpi in cui i documenti sono stati etichettati con le categorie. Utilizzando questi corpi, possiamo costruire classificatori che aggiungono automaticamente nuovi documenti con appropriate etichette di categoria. In primo luogo, noi costruiamo un elenco dei documenti, etichettati con le categorie appropriate. Per questo esempio, abbiamo scelto il corpo dell'esame della pellicola, che categorizza ogni controllo 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)
```

Successivamente, si definisce un estrattore di funzione per i documenti, in modo che il classificatore saprà a quali aspetti dei dati dovrebbe prestare più attenzione (Esempio 6.4). Per l'identificazione dell'argomento del documento, possiamo definire una funzione per ogni parola, indicando se il documento contiene quella parola. Per limitare il numero di caratteristiche che il classificatore deve elaborare, iniziamo con la costruzione di una lista delle 2000 parole più frequenti nel corpo complessivo. Possiamo quindi definire un caratteristico estrattore che semplicemente verifica se ciascuna di queste parole è presente in un dato documento.

```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())

word_features = all_words.keys()[:2000]


def document_features(document):
```

```
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains(%s)' % word] = (word in document_words)
    return features
```

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

Ora che abbiamo definito il nostro estrattore di funzione, lo si può utilizzare per formare un classificatore e etichettare recensioni nuovi film (Esempio 6.5). Per verificare qual è l'affidabilità del classificatore risultante, si calcola la sua precisione sul set di test . E ancora una volta, possiamo usare *show_most_informative_features ()* per scoprire quali caratteristiche del classificatore sono più informative.

```
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

>>> 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
```

A quanto pare in questo corpo, una recensione che parla di "Seagal" è quasi 8 volte più probabile a essere negativa che positiva, mentre una recensione che parla di "Damon" è circa 6 volte più probabile a essere positiva.

Etichette di una parte del discorso

Nel capitolo 5 abbiamo stabilito una piccola espressione regolare che sceglie l'etichetta di una parte del discorso per una parola, cercando all'interno della composizione della parola. Tuttavia, questo piccola espressione regolare doveva essere fatta a mano. Al contrario noi siamo in grado di formare un classificatore per capire quali suffissi sono più informativi. Cominciamo a scoprire quali sono i suffissi più comuni:

```
>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
...     word = word.lower()
...     suffix_fdist.inc(word[-1:])
...     suffix_fdist.inc(word[-2:])
...     suffix_fdist.inc(word[-3:])
```

```
>>> common_suffixes = suffix_fdist.keys()[:100]
>>> print common_suffixes
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the',
 'y', 'r', 'to', 'in', 'f', 'o', 'ed', 'nd', 'is', 'on', 'l',
 'g', 'and', 'ng', 'er', 'as', 'ing', 'h', 'at', 'es', 'or',
 're', 'it', '``', 'an', "''", 'm', ';', 'i', 'ly', 'ion', ...]
```

Poi si definisce una caratteristica di un estrattore di funzione che controlla una determinata parola per questi suffissi:

```
>>> def pos_features(word):
...     features = {}
...     for suffix in common_suffixes:
...         features['endswith(%s)' % suffix] =
word.lower().endswith(suffix)
...     return features
```

L'estrattore di funzione si comporta come gli occhiali scuri, facendo notare alcune delle proprietà (colori) nei nostri dati e rendendo impossibile vedere altre proprietà. Il classificatore si baserà esclusivamente su queste proprietà evidenziate al momento di determinare come etichettare gli input. In questo caso, il classificatore adotterà una decisione basata solo sulle informazioni di suffissi comuni che ha una data parola.

Ora che abbiamo definito la nostra funzione di estrattore, lo possiamo utilizzare per formare un nuovo classificatore "albero della decisione" (che sarà discusso nella Sezione 6.4):

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in tagged_words]
```

```
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
```

```
>>> classifier = nltk.DecisionTreeClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.62705121829935351
```

```
>>> classifier.classify(pos_features('cats'))
'NNS'
```

Una caratteristica piacevole di modelli di "albero delle decisioni" è che sono spesso abbastanza facili da interpretare - possiamo anche istruire NLTK per stamparlo come pseudocodice:

```
>>> print classifier.pseudocode(depth=4)
if endswith(,) == True: return ','
if endswith(,) == False:
  if endswith(the) == True: return 'AT'
  if endswith(the) == False:
    if endswith(s) == True:
      if endswith(is) == True: return 'BEZ'
      if endswith(is) == False: return 'VBZ'
    if endswith(s) == False:
      if endswith(.) == True: return '.'
      if endswith(.) == False: return 'NN'
```

Qui, possiamo vedere che il classificatore inizia controllando se una parola termina con una virgola - se è così, allora riceverà l'aggiunta speciale "," . Successivamente, il classificatore controlla se la parola termina in "la" , in questo caso è quasi certamente un determinatore. Questo "suffisso" si abitua presto alla struttura decisionale perché la parola "il" è comune. Proseguendo, il classificatore controlla se la parola termina in "s". Se è così, allora c'è più probabilità di ricevere l'etichetta del verbo VBZ, e se non, allora è più probabile che sia un sostantivo.

Sfruttare l'ambiente

Migliorando le caratteristiche dell'estrattore di funzione, potremmo modificare questa etichetta della parte del discorso per sfruttare una serie di altri testi, come ad esempio la lunghezza della parola, il numero di sillabe che contiene, o il suo prefisso. Tuttavia, fino a quando l'estrattore di funzione guarda solo alla parola, "aggiunta", non abbiamo modo di aggiungere caratteristiche che dipendono dalla *situazione* in cui la parola compare, Ma caratteristiche contestuali spesso forniscono indizi potenti per l'etichetta corretta - per esempio, quando codifichiamo la parola "volare", sapendo che la parola precedente è "a" ci permetterà di capire che la parola funzioni come un sostantivo, non come un verbo.

Al fine di soddisfare le caratteristiche che dipendono dal contesto di una parola, dobbiamo rivedere il modello che abbiamo usato per definire il nostro estrattore di funzione. Invece di passare alla parola da contrassegnare, si passa a una completa frase, insieme con l'indice della parola etichettata. Questo approccio viene illustrato nell' Esempio 6.6 , che impiega un estrattore di un contesto per definire una parte della classificazione della parola codificata.

```
def pos_features(sentence, i):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
    return features
```

```
>>> pos_features(brown.sents()[0], 8)
{'suffix(3)': 'ion', 'prev-word': 'an', 'suffix(2)': 'on', 'suffix(1)':
'n'}

>>> tagged_sents = brown.tagged_sents(categories='news')
>>> featuresets = []
>>> for tagged_sent in tagged_sents:
...     untagged_sent = nltk.tag.untag(tagged_sent)
...     for i, (word, tag) in enumerate(tagged_sent):
...         featuresets.append( (pos_features(untagged_sent, i), tag) )

>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)

>>> nltk.classify.accuracy(classifier, test_set)
0.78915962207856782
```

E 'chiaro che sfruttando le caratteristiche contestuali le prestazioni della nostra parte del discorso codificato migliorano. Per esempio, il classificatore apprende che una parola è probabile che sia un sostantivo se viene immediatamente dopo il termine "grandi" o la parola "governatoriale". Tuttavia, non è in grado di apprendere che una parola è probabilmente un sostantivo se segue un aggettivo, perché non ha accesso alla precedente etichetta della parte del discorso. In generale, semplici classificatori trattano ogni input come indipendente da tutti gli altri input. In molti contesti, questo ha perfettamente senso. Ad esempio, le decisioni circa se i nomi tendono a essere maschili o femminili possono essere eseguita caso per caso. Tuttavia, ci sono spesso casi, come l'etichetta di una parte del discorso in cui ci interessa la risoluzione dei problemi di classificazione che sono strettamente correlati tra loro.

Classificazione della sequenza

Al fine di cogliere le relazioni tra le relative attività di classificazione, possiamo usare modelli di **classificatori comuni** che scelgono un'etichettatura appropriata per un insieme di fattori correlati. Nel caso della parte del discorso aggiunto, una varietà di differenti modelli di **classificatori di sequenze** possono essere utilizzate per scegliere congiuntamente l'etichetta della parte di un discorso per tutte le parole di una frase data.

Una strategia di classificazione della sequenza, nota come **classificazione consecutiva** è quella di trovare l'etichetta di classe più probabile per il primo input, quindi di utilizzare la risposta per aiutare a trovare la migliore etichetta per l'input successivo. Il processo può essere ripetuto fino a quando tutti gli ingressi sono stati etichettati. Questo è l'approccio che è stata presa dall'etichetta *bigram* nella Sezione 5.5 , che ha avuto inizio con la scelta di una parte del discorso variabile per la prima parola nella frase, e poi ha scelto l'etichetta per ogni parola successiva in base alla parola stessa e la prevista aggiunta per la parola precedente.

Questa strategia è dimostrata nell' Esempio 6.7 . In primo luogo, dobbiamo aumentare il nostro estrattore di funzione e prendere un argomento della storia, che fornisce un elenco delle etichette che abbiamo finora previsto per la sentenza. Ogni etichetta nella storia corrisponde a una parola della frase . Ma nota che la storia conterrà solo le etichette per le parole che abbiamo già classificato, cioè, le parole a sinistra della parola etichettata. Così, mentre è possibile osservare alcune caratteristiche di parole a destra della parola etichettata, non è possibile guardare le etichette per quelle parole (giacchè non li abbiamo ancora generati).

Dopo aver definito un estrattore di funzione, si può costruire il nostro classificatore di sequenza. Durante il lavoro, aggiungiamo annotati a fornire la storia adeguata all'estrattore di funzione, ma quando aggiungiamo nuove frasi, generiamo l'elenco cronologico in base all'uscita della stessa etichetta.

```
def pos_features(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[I][-2:],
                "suffix(3)": sentence[I][-3:]}
    if I == 0:
        features["prev-word"] = "<START>"
        features["prev-tag"] = "<START>"
    else:
```

```
        features["prev-word"] = sentence[i-1]
        features["prev-tag"] = history[i-1]
    return features


class ConsecutivePosTagger(nltk.TaggerI):

    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for I, (word, tag) in enumerate(tagged_sent):
                featureset = pos_features(untagged_sent, I, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.NaiveBayesClassifier.train(train_set)

    def tag(self, sentence):
        history = []
        for I, word in enumerate(sentence):
            featureset = pos_features(sentence, I, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
```

```
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> size = int(len(tagged_sents) * 0.1)
>>> train_sents, test_sents = tagged_sents[size:], tagged_sents[:size]
>>> tagger = ConsecutivePosTagger(train_sents)
>>> print tagger.evaluate(test_sents)
0.79796012981
```

Altri metodi per la classificazione della sequenza

Un difetto di quest'approccio è che ci impegniamo a ogni decisione che prendiamo. Per esempio, se si decide di etichettare una parola “come” un sostantivo, ma in seguito trovare le prove che sarebbe potuto essere un verbo, non c'è modo di tornare indietro e correggere il nostro errore. Una soluzione a questo problema è di adottare invece una strategia di trasformazione. I Classificatori trasformazionali di lavoro congiunto con la creazione di una prima assegnazione di etichette per gli input, e poi iterativamente raffinando l'assegnazione, nel tentativo di riparare incongruenze tra gli input collegati. l’etichetta Brilll descritto nella sezione (1), è un buon esempio di questa strategia.

Un'altra soluzione è quella di assegnare un punteggio a tutte le possibili sequenze di etichette della parte di un discorso è di scegliere la sequenza il cui punteggio complessivo è più alto. Questo è l'approccio adottato dai modelli **Hidden Markov**. Essi sono simili ai classificatori consecutivi, nel senso che guardano entrambi gli imput e la storia delle aggiunte previste. Tuttavia, piuttosto che trovare l’etichetta migliore a una determinata parola, generano una distribuzione di probabilità sull’aggiunta. Queste probabilità sono poi combinate per calcolare i punteggi di probabilità per le sequenze etichettate ed è scelta la sequenza etichettata con maggiore probabilità. Purtroppo, il numero di sequenze possibili etichettate è abbastanza grande. Data un'aggiunta con 30 etichette, ci sono circa 600 miliardi (30^{10}) per etichettare una frase di 10 parole. Al fine di evitare di considerare tutte queste possibili sequenze separatamente, i modelli Hidden Markov richiedono che l’estrattore di funzione deve solo guardare l'etichetta più recente (o le più recenti *n* tag, dove *n* è abbastanza piccola). Questa restrizione, è possibile utilizzarla con una

programmazione dinamica (Sezione 4.7) per trovare in modo efficiente la sequenza aggiunta più probabile. In particolare, per ogni parola consecutiva con indice *i* , il punteggio viene calcolato per ogni aggiunta possibile corrente e precedente. Questo stesso approcci viene prelevato da due modelli più avanzati, chiamato **Maximum Entropy Markov Models** e **Linear-Chain Conditional Random Field.**

6.2 Ulteriori esempi di classificazione supervisionata

Segmentazione della frase

La segmentazione della frase può essere vista come un compito di classificazione per la punteggiatura: ogni volta che incontriamo un simbolo che potrebbe finire una frase, ad esempio un punto o un punto interrogativo, dobbiamo decidere se termina la frase precedente.

Il primo passo è ottenere alcuni dati che sono stati già segmentati in frasi e convertirli in una forma che è adatta per l'estrazione di caratteristiche:

```
>>> sents = nltk.corpus.treebank_raw.sents ()
 >>> tokens = []
 >>> boundaries = set ()
 >>> offiset = 0
 >>> for sent in nltk.corpus.treebank_raw.sents () :
 ...     tokens.extend (sent)
 ...     offiset + = len (sent)
 ...     boundaries.add (offiset-1)
```

Successivamente, è necessario specificare le caratteristiche dei dati che verranno utilizzati al fine di decidere se la punteggiatura indica una frase-limite:

```
>>> def punct_features(tokens, i):
...     return {'next-word-capitalized': tokens[i+1][0].isupper(),
...             'prevword': tokens[i-1].lower(),
...             'punct': tokens[i],
...             'prev-word-is-one-char': len(tokens[i-1]) == 1}
```

Sulla base di questo estrattore di funzione, siamo in grado di creare un elenco di caratteristiche etichettate, selezionando tutti i tokens di punteggiatura, e individuare se sono tokens di confine o meno:

```
>>> featuresets = [(punct_features(tokens, i), (i in boundaries))
...                for i in range(1, len(tokens)-1)
...                if tokens[i] in '.?!']
```

Utilizzando queste caratteristiche, siamo in grado di formare e valutare la punteggiatura di un classificatore:

```
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.97419354838709682
```

Per utilizzare questo classificatore per eseguire la segmentazione frase, ci limitiamo a controllare ogni segno di punteggiatura per vedere se questa è etichettato come un confine, e dividere l'elenco delle parole al confine. Lo schema nell' Esempio 6.8 mostra come questo può essere fatto.

```
def segment_sentences(words):
    start = 0
    sents = []
    for i, word in enumerate(words):
        if word in '.?!' and classifier.classify(punct_features(words,
i)) == True:
            sents.append(words[start:i+1])
            start = i+1
    if start < len(words):
        sents.append(words[start:])
    return sents
```

Identificare le qualità dell' atto del Dialogo

Durante l'elaborazione del dialogo, può essere utile pensare agli enunciati come un tipo di *azione* svolta dall'interlocutore. Questa interpretazione è più semplice per le dichiarazioni performative come "Io ti perdono" o "Scommetto che non puoi salire quella collina." Ma saluti, domande, risposte, asserzioni, e chiarimenti possono essere considerati come i tipi di intervento basati su azioni. Riconoscere **l'atto del dialogo** alla base delle espressione, può essere un primo passo importante nella comprensione della conversazione.

La NPS Chat Corpus, che è stata dimostrata nella Sezione 2.1 , si compone di oltre 10.000 messaggi di sessioni di messaggistica istantanea. Questi messaggi sono stati tutti etichettati con una delle 15 qualità di atto del dialogo, come "Statement", "Emotion", "ynQuestion", e "Continuer". Possiamo quindi utilizzare questi dati per costruire un classificatore in grado di identificare le qualità dell' atto del dialogo per i nuovi messaggi di messaggistica istantanea. Il primo passo è estrarre i dati di messaggistica di base. Chiameremo *xml_posts ()* per ottenere una struttura di dati che rappresenta l'annotazione XML per ogni post:

```
>>> posts nltk.corpus.nps_chat.xml_posts = () [: 10000]
```

Poi si definisce un semplice estrattore di funzione che controlla quali parole contiene il post:

```
>>> def dialogue_act_features(post):
...     features = {}
...     for word in nltk.word_tokenize(post):
...         features['contains(%s)' % word.lower()] = True
...     return features
```

Infine, noi costruiamo i dati di lavoro e di test applicando l'estrattore di funzione per ogni post *(post.get ('class'),* e creare un nuovo classificatore:

```
>>> featuresets = [(dialogue_act_features(post.text), post.get('class'))
...                 for post in posts]
```

```
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.66
```

Riconoscere l'implicazione testuale

Riconoscere l'implicazione testuale (RTE) è il compito di stabilire se un dato pezzo di testo *T* comporta un altro testo chiamato "*hypothesis*" (come già discusso nella Sezione 1.5). Qui ci sono un paio di esempi di testo / ipotesi. L'etichetta *vero* indica che l'implicazione regge, e *Falso* , che non riesce a reggere.

Sfida 3, coppia 34 (True)

T : Parviz Davudi rappresentava l'Iran in una riunione della cooperazione di Shanghai (SCO), l'associazione neonata che lega la Russia, la Cina e quattro repubbliche ex sovietiche dell'Asia centrale insieme per combattere il terrorismo.

H : La Cina è un membro della SCO.

Sfida 3, coppia 81 (False)

T : Ai sensi degli articoli NC di organizzazione, i membri della società LLC sono H. Nelson Beavers, III, H. Chester Beavers e Jennie Beavers Stewart.

H : Jennie Beavers Stewart è un azionista di Carolina Analytical Laboratory.

Va sottolineato che il rapporto tra testo e ipotesi non è destinato ad essere un implicazione logica, ma un umano potrebbe concludere che il testo fornisce prova ragionevole per prendere l'ipotesi come vera.

Siamo in grado di trattare RTE come un compito di classificazione, in cui si cerca di prevedere il *Vero* / *Falso* dell'etichetta per ogni coppia. Anche se sembra probabile che gli approcci di successo per questo compito comportano una combinazione di analisi semantica e la conoscenza del mondo reale, molti primi tentativi di RTE hanno ottenuto risultati abbastanza buoni con l'analisi superficiale, in base alla somiglianza tra il testo e ipotesi a livello di parola. Nel caso ideale, ci si aspetterebbe che se c'è un assegnazione allora tutte le informazioni espresse dall'ipotesi dovrebbero essere presenti anche nel testo.

Nel nostro rivelatore di funzione di RTE (Esempio 6.9), lasciamo che le parole (ad esempio, le qualità delle parole) lavorino come indici di informazioni, e le nostre caratteristiche contino il grado di sovrapposizione di parole e il grado in cui ci sono ipotesi di parole, ma non nel testo. Non tutte le parole sono ugualmente importanti – *Named Entity* accenna ad esempio ai nomi di persone, organizzazioni e luoghi che sono probabilmente più significativi, ciò ci spinge a estrarre le distinte formazioni sulla *word***s** e *ne***s** (Named Entities).

```
def  rte_features (rtepair):
    extractor = nltk.RTEFeatureExtractor (rtepair)
```

```
    features = {}
    features [ 'word_overlap' ] = len (extractor.overlap ( 'word' ))
    features [ 'word_hyp_extra' ] = len (extractor.hyp_extra ( 'word' ))
    features [ 'ne_overlap' ] = len (extractor.overlap ( 'ne' ))
    features [ 'ne_hyp_extra' ] = len (extractor.hyp_extra ( 'ne' ))
    return features
```

Per illustrare il contenuto di queste caratteristiche, esaminiamo alcuni attributi del testo / ipotesi Coppia 34 mostrato in precedenza:

```
>>> rtepair = nltk.corpus.rte.pairs(['rte3_dev.xml'])[33]
>>> extractor = nltk.RTEFeatureExtractor(rtepair)
>>> print extractor.text_words
set(['Russia', 'Organisation', 'Shanghai', 'Asia', 'four', 'at',
'operation', 'SCO', ...])
>>> print extractor.hyp_words
set(['member', 'SCO', 'China'])
>>> print extractor.overlap('word')
set([])
>>> print extractor.overlap('ne')
set(['SCO', 'China'])
>>> print extractor.hyp_extra('word')
set(['member'])
```

Queste caratteristiche indicano che tutte le parole importanti nella ipotesi sono contenute nel testo, e quindi ciò si può etichettare come *vero* .

Il modulo *nltk.classify.rte_classify* raggiunge poco più del 58% di accuratezza dei dati combinati RTE con metodi come questi. Anche se questa cifra non è molto impressionante, richiede uno sforzo notevole, di elaborazione linguistica, per ottenere risultati molto migliori.

Aumentare maggiormente l'insieme di dati

Python fornisce un ambiente eccellente per eseguire l'elaborazione di testo di base e di estrazione di caratteristiche. Se si tenta di utilizzare Python su insiemi di dati di grandi dimensioni, è possibile che l'algoritmo di apprendimento richiede una quantità irragionevole di tempo e memoria per completare.

Se si prevede di formare classificatori con grandi quantità di dati di lavoro o di un gran numero di caratteristiche, si consiglia di esplorare le strutture di NLTK per l'interfacciamento con i pacchetti esterni di apprendimento automatico. Una volta che questi pacchetti sono stati installati, NLTK può chiamarli (tramite chiamate di sistema) per formare i modelli di classificazione molto più velocemente rispetto alle inculcazioni della classificazione di Python .

6.3 Valutazione

Per decidere se un modello di classificazione sia accurato per catturare un modello, dobbiamo valutare quel modello. Il risultato di questa valutazione è importante per decidere come il modello è affidabile, e per quali scopi si può utilizzare. La valutazione può anche essere uno strumento efficace per averci guidato nel migliorare il modello.

Il Test Set

La maggior parte delle tecniche di valutazione calcolano un punteggio per un modello confrontando le etichette che esso genera per gli input in un **insieme di test** (**set di valutazione**), con le etichette corrette per tali input. Questo insieme di test ha tipicamente lo stesso formato del set di lavoro. Tuttavia, è molto importante che l'insieme di test sia distinto dal corpo di formazione: se riutilizziamo il set di lavoro semplicemente, come set di test, quindi un modello che semplicemente memorizza il suo input, potrebbe ricevere punteggi erroneamente alti.

Quando si costruisce il set di prova, vi è spesso un compromesso tra la quantità di dati disponibili per i test e l'importo disponibile per la formazione. Per classificare i compiti che hanno un piccolo numero di ben bilanciate etichette e di test diversificata, una significativa valutazione può essere eseguita con solo 100 casi di valutazione. Ma se un compito di classificazione ha un grande numero di etichette, o comprende etichette molto frequenti, allora la dimensione del set di test deve essere scelta in modo che l'etichetta meno frequente avviene almeno 50 volte. Inoltre, se l'insieme di test contiene molti casi strettamente correlati - quali i casi tratti da un unico documento - allora la dimensione del set di prova deve essere aumentato per garantire che questa mancanza non sbagli i risultati della valutazione. Quando grandi quantità di dati annotati sono disponibili, è comune sbagliare sul lato di sicurezza utilizzando il 10% di tutti i dati per la valutazione.

Un'altra considerazione nella scelta del set di prova è il grado di somiglianza tra istanze presenti nel set di test e quelli del set di sviluppo. Si consideri ad esempio il compito della parte del discorso aggiunto. A un estremo, si potrebbe creare il set di lavoro e di test assegnando frasi da una fonte di dati che riflette un solo genere:

```
>>> import random
>>> from nltk.corpus import brown
>>> tagged_sents = list(brown.tagged_sents(categories='news'))
>>> random.shuffle(tagged_sents)
>>> size = int(len(tagged_sents) * 0.1)
>>> train_set, test_set = tagged_sents[size:], tagged_sents[:size]
```

In questo caso, il nostro set di test sarà *molto* simile al nostro set di lavoro. Il set di lavoro e il set di prova sono prese dallo stesso genere, e quindi non possiamo essere sicuri che i risultati della valutazione potrebbero generalizzare altri generi. Quel che è peggio, a causa della chiamata *a random.shuffle ()* , il set di test contiene frasi che sono prese dagli stessi documenti che sono stati utilizzati per la formazione. Se c'è un modello coerente all'interno di un documento - per esempio, se una determinata parola appare spesso insieme ad una particolare parte del discorso aggiunta - allora tale differenza rifletterà sia l'insieme di sviluppo che di test. Un approccio leggermente migliore è assicurare che il set di lavoro e il set di test sono presi da diversi documenti:

```
>>> file_ids = brown.fileids(categories='news')
>>> size = int(len(file_ids) * 0.1)
>>> train_set = brown.tagged_sents(file_ids[size:])
>>> test_set = brown.tagged_sents(file_ids[:size])
```

Se si desidera eseguire una valutazione più rigorosa, siamo in grado di disegnare il set test da documenti che sono meno strettamente correlati a quelli del set di lavoro:

```
>>> train_set = brown.tagged_sents(categories='news')
>>> test_set = brown.tagged_sents(categories='fiction')
```

Se costruiamo un classificatore che si comporta bene su questa serie di test, allora possiamo essere certi che essa ha il potere di generalizzare bene.

Precisione

La metrica semplice che può essere utilizzata per valutare un classificatore, la **precisione** , misura la percentuale di input nel test che il classificatore ha correttamente etichettato. Ad esempio, un classificatore del sesso del nome che predice il nome corretto 60 volte in un insieme di test contenente 80 nomi avrebbe una precisione di 60/80 = 75%. La funzione *nltk.classify.accuracy ()* calcola l'accuratezza di un modello di classificatore su un insieme del test dato:

```
>>> classifier = nltk.NaiveBayesClassifier.train (train_set)
 >>> print  'Accuracy: 4.2f%' nltk.classify.accuracy% (classifier,
test_set)
 0,75
```

Nell'interpretare il punteggio di un classificatore, è importante prendere in considerazione le frequenze delle etichette di classe individuali nel set di test. Ad esempio, si consideri un classificatore che determina il senso corretto di parole per ogni occorrenza della parola *bank*. Se valutiamo questo classificatore sul testo della finanziaria, allora potremmo scoprire che il senso dell' *istituzione- finanziaria* appare 19 volte su 20. In questo caso, una precisione del 95% sarebbe poco impressionante, dato che abbiamo potuto ottenere la precisione con un modello che restituisce sempre il senso dell' istituto finanziario. Tuttavia, se invece valutiamo il classificatore su un corpo più equilibrato, dove il senso della parola più frequente ha una frequenza del 40%, quindi un punteggio precisi del 95% , sarebbe un risultato molto più positivo.

Precisione e revocazione

Un altro caso in cui i conti di precisione possono essere incompresi è nell'attività di "ricerca", come il reperimento delle informazioni, dove stiamo cercando di trovare i documenti che sono rilevanti per una determinata attività. Dal momento che il numero di documenti irrilevanti supera di gran lunga il numero di documenti pertinenti, il punteggio di accuratezza per un modello che etichetta tutti i documenti come irrilevanti sarebbe molto vicino al 100%.

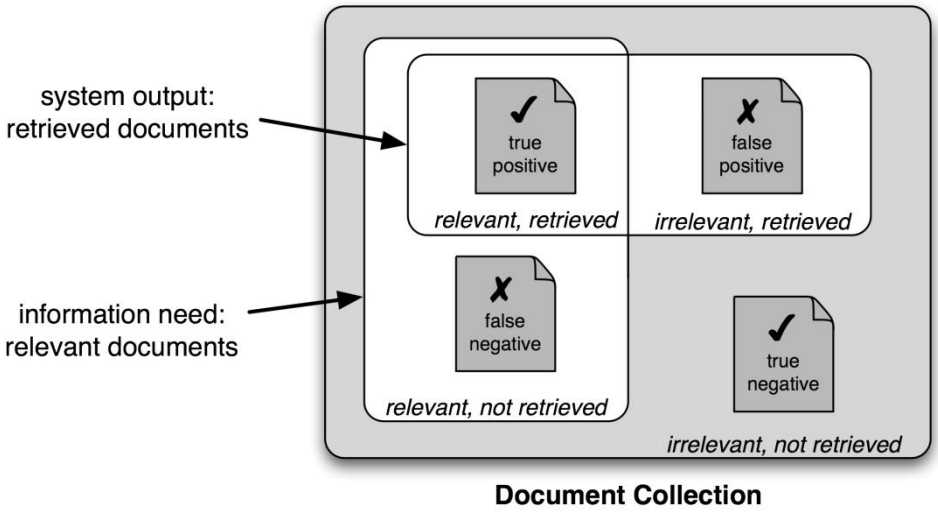


Figura 6.10 : veri e falsi positivi e le negazioni vere e false.

È pertanto convenzionale impiegare un diverso insieme di misure per compiti di ricerca, in base al numero di elementi in ciascuna delle quattro categorie indicate in Figura 6.10 :

- **Veri positivi** sono elementi rilevanti che abbiamo correttamente individuati come rilevanti.
- **Veri negativi** sono elementi irrilevanti che sono correttamente identificati come irrilevanti.
- **I falsi positivi** (o **errori di tipo I**) sono elementi irrilevanti che noi erroneamente abbiamo identificato come rilevanti.
- **I falsi negativi** (o **errori di di tipo II**) sono elementi rilevanti che abbiamo erroneamente individuato come irrilevanti.

Alla luce di questi quattro numeri, possiamo definire i seguenti parametri:

- **Di precisione** , che indica come molti degli elementi che abbiamo identificato erano pertinenti, è *TP / (TP + FP)* .
- **Di richiamo** , che indica come molti degli elementi rilevanti che abbiamo identificato, è *TP / (TP + FN)* .
- L' **F-Measure** (o **F-Score**), che combina la precisione e da un punteggio unico, è definito come la media armonica della precisione e richiamo: (2 × *precisione* × *Recall*) / (*precisione* + *Recall*).

Matrice confusa

Quando si eseguono i compiti di classificazione con tre o più etichette, può essere informativo per suddividere gli errori commessi dal modello basato su alcune qualità di errori fatti. Una **matrice di confusione** è una tabella in cui ogni cella [*i* , *j*] indica la frequenza dell'etichetta *j* che era stato previsto quando l'etichetta corretta era *i* .Pertanto, gli elementi diagonali (cioè, le celle [*i* , *j*]) indicano le etichette che sono state correttamente previste, e fuori dalla diagonale le voci indicano gli errori. Nell'esempio che segue, si genera una matrice di confusione per un'unica aggiunta, sviluppato in Sezione 5.4

```
>>> def tag_list(tagged_sents):
...     return [tag for sent in tagged_sents for (word, tag) in sent]
>>> def apply_tagger(tagger, corpus):
...     return [tagger.tag(nltk.tag.untag(sent)) for sent in corpus]
>>> gold = tag_list(brown.tagged_sents(categories='editorial'))
>>> test = tag_list(apply_tagger(t2,
brown.tagged_sents(categories='editorial')))
>>> cm = nltk.ConfusionMatrix(gold, test)
>>> print cm.pp(sort_by_count=True, show_percents=True, truncate=9)
    |                                         N                       |
    |      N      I      A      J             N             V      N  |
    |      N      N      T      J      .      S      ,      B      P  |
----+-----------------------------------------------------------------+
 NN | <11.8%>  0.0%      .   0.2%      .   0.0%      .   0.3%   0.0%  |
 IN |   0.0%  <9.0%>     .      .      .   0.0%      .      .      .  |
 AT |      .      .  <8.6%>     .      .      .      .      .      .  |
 JJ |   1.6%      .      .  <4.0%>     .      .      .   0.0%   0.0%  |
  . |      .      .      .      .  <4.8%>     .      .      .      .  |
NNS |   1.5%      .      .      .      .  <3.2%>     .      .   0.0%  |
  , |      .      .      .      .      .      .  <4.4%>     .      .  |
 VB |   0.9%      .      .   0.0%      .      .      .  <2.4%>     .  |
 NP |   1.0%      .      .   0.0%      .      .      .      .  <1.9%>|
----+-----------------------------------------------------------------+
(row = reference; col = test)
```

La matrice di confusione indica che errori comuni includono una sostituzione di NN per JJ (per 1,6% di parole), e di NN per NNS (per 1,5% di parole). Si noti che i punti, che corrispondono alle classificazioni corrette - sono contrassegnati con parentesi angolari.

Convalida incrociata

Per valutare i nostri modelli, dobbiamo riservare una parte dei dati di set di test. Come abbiamo già detto, se l'insieme di test è troppo piccola, allora la nostra valutazione non può essere precisa. Tuttavia, fare la prova più grande del set di test di solito significa rendere il set di lavoro più piccolo, che può avere un impatto significativo sulle prestazioni. Una soluzione a questo problema è quello di effettuare valutazioni multiple su diversi set di test, quindi combinare i risultati di tali valutazioni, tecnica nota come **cross-validation** . In particolare, suddividere il corpo originale in *N* sottoinsiemi chiamati **pieghe** . Per ognuna di queste pieghe, formiamo un modello utilizzando tutti i dati , *tranne* i dati che si piegano, e quindi verificare questo modello sulla piega. Anche se le singole pieghe potrebbero essere troppo piccole per dare punteggi con un' accurata valutazione, il punteggio di valutazione combinato è basato su una grande quantità di dati, e quindi è abbastanza affidabile.

Un secondo e altrettanto importante vantaggio di utilizzare la convalida incrociata è che ci permette di esaminare in che modo ampiamente le varie prestazioni accostano differenti set di lavoro. Se riusciamo ad ottenere risultati molto simili per tutti i set di lavoro *N*, allora possiamo essere ragionevolmente sicuri che il punteggio è accurato. D'altra parte, se i punteggi variano ampiamente tra i set di lavoro N, allora probabilmente dovremmo essere scettici circa l'esattezza del punteggio di valutazione.

6.4 Alberi di decisione

Nelle prossime tre sezioni, daremo uno sguardo più da vicino a 3 metodi di apprendimento che possono essere utilizzati per costruire automaticamente modelli di classificazione: alberi decisionali, classificatori naive Bayes e classificatori di massima entropia. Come abbiamo visto, è possibile trattare questi metodi di apprendimento come scatole nere, semplicemente formando modelli e che li utilizzano per la predizione senza capire come funzionano. La comprensione di questi metodi possono aiutare a guidare la nostra selezione di caratteristiche adeguate, e soprattutto le nostre decisioni su come tali caratteristiche devono essere codificate. E la comprensione dei modelli generati ci permette di estrarre le informazioni su quali caratteristiche sono più importanti, e come, tali caratteristiche si relazionano tra loro.

Un **albero di decisione** è un semplice diagramma di flusso che seleziona etichette per valori di input. Questo diagramma di flusso è costituito da **nodi decisionali**, che controllano i valori di funzionalità, e **nodi foglia** , che assegnano etichette. Per scegliere l'etichetta per un valore di input, cominciamo dal nodo decisionale iniziale del diagramma di flusso, noto come il suo **nodo principale** . Questo nodo contiene una condizione che verifica una delle caratteristiche del valore di input, e seleziona un ramo sulla base del valore della funzionalità stessa. In seguito il ramo che descrive il nostro valore di input, ci fa arrivare ad un nuovo nodo decisionale, con una nuova condizione sulle caratteristiche del valore di input. Continuiamo seguendo il ramo selezionato in base alle condizioni di ciascun nodo, fino ad arrivare ad un nodo foglia che prevede un'etichetta per il valore di input. La Figura 6.11 mostra un esempio di modello di decisione ad albero per l'attività di sesso del nome.

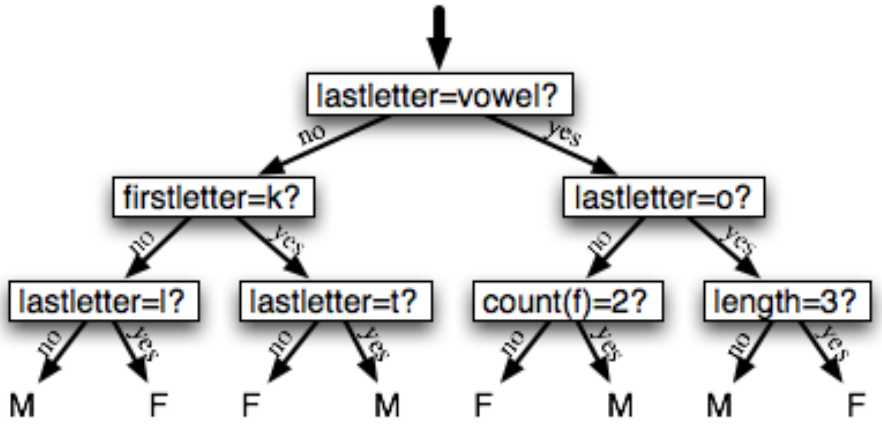


Figura 6.11 : Modello Albero decisionale per l'attività di sesso del nome. Si noti che diagrammi ad albero sono convenzionalmente stabiliti "a testa in giù", con la radice in alto e le foglie in basso.

Una volta che abbiamo un albero di decisione, è semplice da utilizzare per assegnare etichette ai valori di nuovi input. Quello che è meno chiaro è come si può costruire un albero di decisione che modella un dato set di lavoro. Ma prima di guardare all'algoritmo di apprendimento per la costruzione di alberi di decisione, considereremo un compito semplice: scegliere la migliore "matrice di decisione" per un corpo. Un **ceppo di decisione** è un albero decisionale con un singolo nodo che decide come classificare gli input sulla base di una singola caratteristica. Esso contiene una foglia per ogni valore, specificando l'etichetta di classe che devono essere assegnati agli input le cui

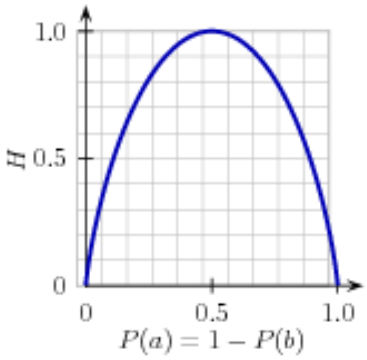
caratteristiche hanno quel valore. Al fine di costruire un ceppo decisione, dobbiamo prima decidere quale funzione deve essere utilizzata. Il metodo più semplice è quello di costruire solo una matrice di decisione per ogni caratteristica possibile, e vedere chi è che raggiunge la massima precisione sui dati di formazione, anche se ci sono altre alternative che discuteremo qui di seguito. Una volta che abbiamo scelto una funzione, possiamo costruire la matrice decisionale assegnando un'etichetta a ogni foglia in base all'etichetta più frequente per gli esempi selezionati nel set di lavoro (ad esempio, i casi in cui la funzione selezionata ha quel valore).

Un dato algoritmo per scegliere matrici di decisioni, l'algoritmo per la crescita di alberi di decisione più grandi è semplice. Iniziamo selezionando il complessivo ceppo di decisione, migliore per l'attività di classificazione. Dobbiamo poi verificare l'esattezza di ciascuna delle foglie sul set di lavoro. Foglie che non ottengono una sufficiente precisione vengono poi sostituiti da nuove matrici di decisione, formate dal sottoinsieme del corpo di lavoro che è selezionato per il percorso della foglia. Per esempio, si potrebbe far crescere l'albero decisionale nella figura 6.11 , sostituendo la foglia più a sinistra, con una nuova matrice di decisione, lavorando sul sottoinsieme dei nomi del set di lavoro che non iniziano con una "k" o terminano con una vocale o una "L . "

Entropia e informazioni di guadagno

Come è stato detto prima, ci sono diversi metodi per identificare la caratteristica più importante per una matrice di decisione. Un'alternativa popolare, chiamata **guadagno di informazioni** , misura quanto diventano più organizzati i valori di input quando li divide con una data caratteristica. Per misurare quanto disorganizzata la serie originale di valori di input sono, si calcola l'entropia delle loro etichette, che sarà alta se i valori di input hanno etichette molto varie, e bassa se i valori di input hanno tutti la stessa etichetta. In particolare, l'entropia è definita come la somma della probabilità di ciascun etichetta per la probabilità di log della stessa etichetta:

(1) $H = -\Sigma_{l \mid a \mid \text{etichette}} P(l) \times \log_2 P(l)$.



Per esempio, Figura 6.12 mostra come l'entropia di etichette nell'attività della predizione nome sesso, dipende dal rapporto del maschio con i nomi femminili. Si noti che se la maggior parte dei valori di input hanno la stessa etichetta (ad esempio, se P (maschio) è vicino a 0 o vicino a 1), l'entropia è bassa. In particolare, le etichette che hanno bassa frequenza non contribuiscono molto alla entropia (poiché *P (l)* è piccolo) e le etichette ad alta frequenza anche non contribuiscono molto alla entropia (dal $\log_2 P(l)$ è

piccola). D'altra parte, se i valori di ingresso hanno una grande varietà di etichette, poi ci sono molte etichette con una frequenza "media", dove né *P (l)* né *log* $_2$ *P (l)* sono piccole, quindi l'entropia è alta. Esempio 6,13 viene illustrato come calcolare l'entropia da un elenco di etichette.

```
import math
def entropy(labels):
    freqdist = nltk.FreqDist(labels)
    probs = [freqdist.freq(l) for l in nltk.FreqDist(labels)]
    return -sum([p * math.log(p,2) for p in probs])
```

```
>>> print entropy(['male', 'male', 'male', 'male'])
0.0
>>> print entropy(['male', 'female', 'male', 'male'])
0.811278124459
>>> print entropy(['female', 'male', 'female', 'male'])
1.0
>>> print entropy(['female', 'female', 'male', 'female'])
0.811278124459
>>> print entropy(['female', 'female', 'female', 'female'])
0.0
```

Una volta che abbiamo calcolato l'entropia del set originale di etichette dei valori di input, siamo in grado di determinare quanto sono più organizzate le etichette e come diventano una volta che si applica la matrice di decisione. Per fare ciò, si calcola l'entropia per ogni ceppo di decisione di foglie , e fare la media dei valori di entropia della foglia (ponderata per il numero di campioni in ogni foglia). Il guadagno di informazioni è quindi pari all' entropia originale meno questa nuova, entropia ridotta. Più alto è il guadagno di informazione,il migliore lavoro della matrice di decisione è di dividere i valori di input in gruppi coerenti, in modo da poter costruire gli alberi di decisione, selezionando i ceppi di decisione con il guadagno più alto di informazioni.

Un'altra considerazione per gli alberi di decisione è l'efficienza. Il semplice algoritmo per la selezione di ceppi di decisione di cui sopra devono costruire un ceppo decisione candidato per ogni caratteristica possibile, e questo processo deve essere ripetuto per ogni nodo nell'albero decisionale costruito. Un certo numero di algoritmi sono stati sviluppati per ridurre il tempo di formazione per la memorizzazione e il riutilizzo delle informazioni su esempi esaminati in precedenza.

Gli alberi decisionali hanno una serie di caratteristiche utili. Tanto per cominciare, sono semplici da capire e facile da interpretare. Ciò è particolarmente vero nella parte superiore dell'albero decisionale, dove di solito è possibile capire l'algoritmo per trovare caratteristiche molto utili. Gli alberi decisionali sono particolarmente adatti ai casi in cui molti distinzioni gerarchiche categoriali possono essere fatte. Ad esempio, gli alberi di decisione possono essere molto efficace nel catturare alberi filogenesi.

Tuttavia, alberi di decisione hanno anche alcuni svantaggi. Un problema è che, dal momento che ogni ramo nell'albero decisionale suddivide i dati di lavoro, la quantità di dati di lavoro a disposizione per la formazione dei nodi inferiori nella struttura può diventare molto piccolo. Come risultato, questi nodi decisionali possono **abbassare** il set di lavoro Una soluzione a questo problema è quello di smettere di dividere i nodi ogni volta che la quantità di dati di formazione diventa troppo piccola. Un'altra soluzione è quella di crescere un albero di decisione, ma poi **potare** i nodi decisionali che non migliorano le prestazioni di un dev-test.

Nella classificazione dei documenti in argomenti, caratteristiche come *hasword (calcio)* sono altamente indicativi di una specifica etichetta, indipendentemente da altri valori di funzione. Poiché non vi è spazio limitato nella parte superiore dell'albero decisionale, la maggior parte di queste caratteristiche dovrà essere ripetuta su molti diversi rami dell'albero. E dal momento che il numero di sportelli aumenta in modo esponenziale man mano che procediamo verso il basso, il numero di ripetizioni può essere molto grande.

Un problema correlato è che gli alberi di decisione non sono bravi a fare uso di funzioni che sono deboli predittori dell'etichetta corretta. Dal momento che queste caratteristiche incrementano miglioramenti relativamente piccoli, tendono a verificarsi molto bassi nell'albero decisionale. Ma quando l'interprete dell'albero di decisione è sceso abbastanza per utilizzare queste funzionalità, non ci sono abbastanza dati di lavoro per determinare in modo affidabile l'effetto che dovrebbero avere. Se potessimo invece osservare l'effetto di queste caratteristiche in tutto l'intero set di lavoro, allora potremmo essere in grado di trarre alcune conclusioni su come dovrebbero influenzare la scelta del marchio.

Il fatto che gli alberi di decisione, richiedono caratteristiche di essere controllata in un ordine specifico, limita la loro capacità di sfruttare le caratteristiche che sono relativamente indipendenti l'uno dall'altro. Il metodo di classificazione di Naive Bayes , di cui parleremo dopo, supera questa limitazione, consentendo tutte le funzioni di agire "in parallelo".

6.5 Classificatori Naive Bayes

Nei classificatori Naive Bayes, ogni funzione ottiene una voce in capitolo nel determinare quale etichetta deve essere assegnata a un valore di input. Per scegliere un'etichetta per un valore di input, il classificatore naive Bayes inizia calcolando la **probabilità a priori** di ciascuna etichetta, che è determinata controllando la frequenza di ciascuna etichetta nel set di lavoro. Il contributo di ciascun elemento viene poi combinato con questa probabilità a priori, per arrivare a una stima di probabilità per ciascuna etichetta.

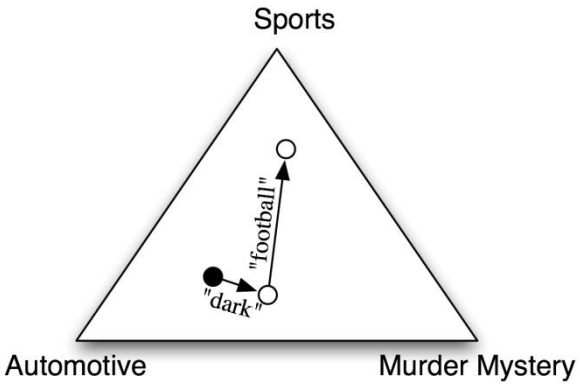
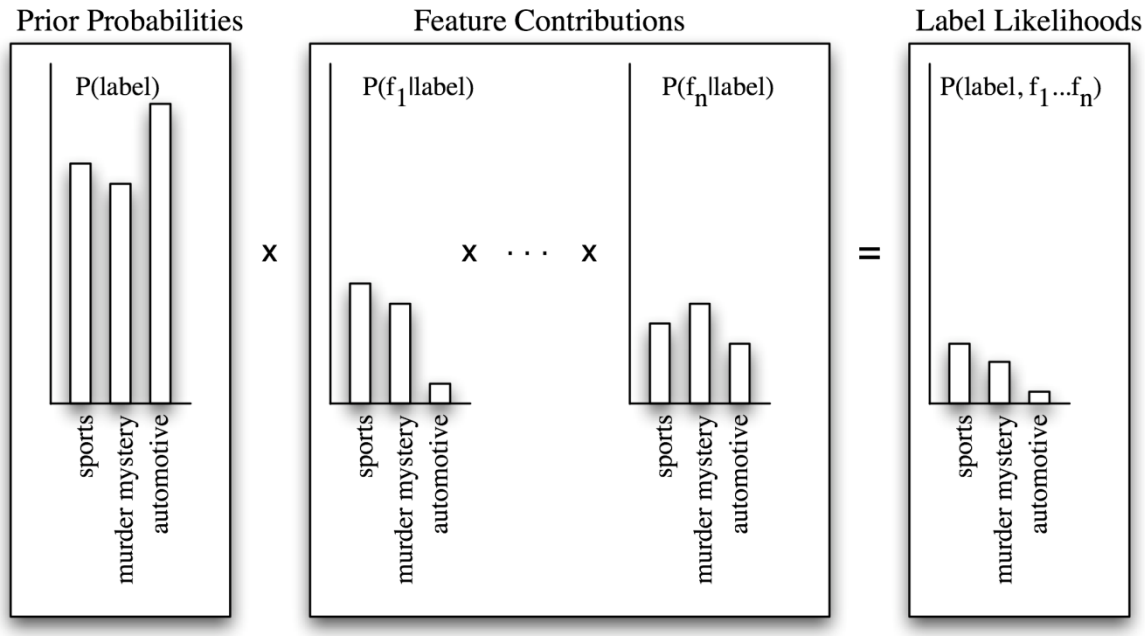


Figura 6.14 : Un esempio astratto della procedura utilizzata dal classificatore Naive Bayes di scegliere l'argomento di un documento. Nel corpo di formazione, la maggior parte dei documenti sono *automobilistici* così il classificatore inizia in un punto più vicino alla etichetta "automotive". Ma considera allora l'effetto di ciascuna funzione. In questo esempio, il documento di input contiene la parola "dark", che è un indicatore debole di romanzi gialli, ma contiene anche il "calcio", parola che è un forte indicatore per i documenti sportivi.

In particolare, il punteggio di verosimiglianza per ogni etichetta viene ridotto moltiplicandolo per la probabilità che avrebbe la funzione di un valore di ingresso con quella etichetta. Ad esempio, se la parola *di esecuzione* si verifica nel 12% dei documenti sportivi, il 10% dei documenti di mistero e il 2% dei documenti automobilistici, il punteggio di probabilità dell' l'etichetta sport sarà moltiplicato per 0,12; il punteggio di probabilità dell'etichetta mistero sarà moltiplicato per 0,1, e il punteggio probabilità dell' etichetta automobilistica sarà moltiplicato per 0,02. L'effetto complessivo sarà di ridurre il punteggio dell'etichetta mistero che è leggermente superiore al punteggio dell'etichetta sport, e di ridurre significativamente l'etichetta automobilistico rispetto alle altre due etichette. Questo processo è illustrato in Figura 6.15 e Figura 6.16



Alla base del modello probabilistico

Un altro modo per comprendere il classificatore naive Bayes è sceglie l'etichetta più probabile per un input, sotto l'ipotesi che ogni valore di ingresso è generato dalla scelta di una classe di etichetta per quel valore di input, e quindi generando ogni caratteristica, completamente indipendente da tutte le altre. Naturalmente, questo presupposto è illusorio; caratteristiche sono spesso altamente interdipendenti. Torneremo ad alcune delle conseguenze di questo assunto, alla fine di questa sezione. Questa semplificazione dell'ipotesi, noto come l' **assunzione di Naive Bayes** (o **assunzione di indipendenza**) combina molto più facilmente i contributi delle diverse caratteristiche, dal momento che non c'è bisogno di preoccuparsi su come devono interagire tra di loro.

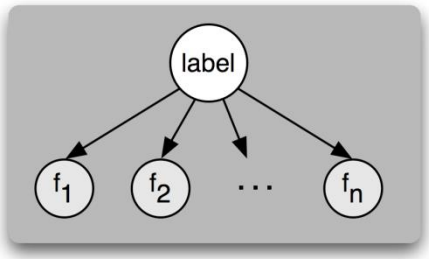


Figura 6.16 : Un **grafico di rete bayesiana** che illustra il processo generativo che viene assunto dal classificatore Naive Bayes. Per generare un input denominato, il primo modello sceglie un'etichetta per l'input, quindi genera ciascuna delle caratteristiche dell'input basata su tale etichetta. Ogni caratteristica si presume essere del tutto indipendente da ogni altra caratteristica.

Sulla base di questa ipotesi, possiamo calcolare un'espressione per *P (etichette | caratteristiche)* , la probabilità che un ingresso avrà un'etichetta particolare dato che ha un particolare insieme di caratteristiche. Per scegliere un'etichetta per un nuovo ingresso, possiamo semplicemente scegliere l'etichetta *l* che massimizza *P (l | caratteristiche)* .

Per iniziare, notiamo che *P (etichette | caratteristiche)* è uguale alla probabilità che un ingresso ha una particolare etichetta *e* l'insieme specificato di caratteristiche, divisa per la probabilità che ha il gruppo specificato di caratteristiche:

(2) *P (etichette | Voti) = P (caratteristiche, etichetta) / P (caratteristiche)*

Successivamente, si nota che *P (caratteristiche)* sarà lo stesso per ogni scelta di etichetta, quindi se ci siamo semplicemente interessati a trovare l'etichetta più probabile, è sufficiente calcolare *P (caratteristiche, etichetta)* , che chiameremo, probabilità di etichetta.

La probabilità di etichetta può essere ampliata come la probabilità dei tempi di etichetta e la probabilità delle caratteristiche di una data etichetta:

(4) *P (caratteristiche, etichetta) = P (etichetta) × P (caratteristiche | etichetta)*

Inoltre, poiché le caratteristiche sono tutte indipendenti tra loro (data l'etichetta), si può separare la probabilità di ogni singola caratteristica:

(5) *P (caratteristiche, etichetta) = P (etichetta)* × Prod $_{\text{f a | caratteristiche di}}$ *P (f | etichetta)* `

Questo è esattamente l'equazione che abbiamo discusso sopra per calcolare la probabilità dell' etichetta: *P (etichetta)* è la probabilità a priori per una determinata etichetta, e ogni *P (f | etichetta)* è il contributo di un singola caratteristica alla probabilità dell' etichetta.

Zero conti e regolarità

Il modo più semplice per calcolare *P (f | etichetta)* , il contributo di una funzione *f* verso la probabilità di etichetta per l'etichetta dell' *etichetta* , è quello di prendere la percentuale di casi di formazione con l'etichetta data visto che ha anche la caratteristica data:

(6) *P (f | etichetta) = count (f, etichetta) / count (etichetta)*

Tuttavia, questo semplice approccio può diventare problematico quando una funzione *non* si verifica con una determinata etichetta del set di lavoro. In questo caso, il nostro valore calcolato per *P (f | label)* sarà zero, questo causerà la probabilità di etichetta per l'etichetta data ad essere zero. Qui Il problema fondamentale è nel nostro calcolo di *P (f | label)* , la probabilità che un ingresso avrà una caratteristica, data una etichetta. In particolare, proprio perché non abbiamo visto una combinazione etichetta/caratteristica che si verifica nel set di lavoro, non significa che non sia impossibile che tale combinazione si verifichi. Ad esempio, non possiamo aver visto tutti i documenti di

omicidio e mistero che contengono la parola "calcio", ma non vogliamo concludere che è completamente impossibile che questi documenti esistono.

Così, sebbene *count(f, label) / count (etichetta)* è una buona stima per *P (f | label)* quando *count (f, label)* è relativamente elevata, tale stima diventa meno affidabile quando *count (f)* diventa più piccolo. Pertanto, quando si costruiscono modelli Naive Bayes, di solito impiegano tecniche più sofisticate, noti come **tecniche regolari** per il calcolo *P (f | label)* , la probabilità di una caratteristica di una data etichetta. Per esempio, la **probabilità di giudizio previsto** per la probabilità di una caratteristica di un'etichetta aggiunge sostanzialmente 0,5 a ciascun *conteggio (f, label)* , e il **giudizio Heldout** utilizza un corpo heldout per calcolare il rapporto tra le frequenze tecniche e la probabilità di crteristiche. Il modulo *nltk.probability* fornisce il supporto per una vasta gamma di tecniche di regolarità.

Funzioni non binarie

Noi qui affermiamo che ogni funzione è binaria, vale a dire che ogni input ha o non ha una funzione. Le caratteristiche dell'etichetta valutata (ad esempio, un colore che può essere rosso, verde, blu, bianco o arancione) possono essere convertite in funzioni binarie sostituendole con funzioni binarie come "colore è rosso". Funzioni numerici possono essere convertite in funzione binarie dal recipiente , che li sostituisce con funzioni come "4 <x <6".

Un'altra alternativa è quella di utilizzare i metodi di regressione per modellare le probabilità di funzioni numeriche. Per esempio, se si assume che la funzione di altezza ha una distribuzione curva a campana, allora potremmo stimare P (altezza | etichetta), trovando la media e la varietà delle altezze degli input con ogni etichetta. In questo caso, *P (f = v | label)* non sarebbe un valore fisso, ma varia a seconda del valore di *v* .

L'ingenuità dell'indipendenza

La ragione per cui i classificatori naive Bayes sono chiamati "ingenui" è che è irragionevole supporre che tutte le funzioni sono indipendenti l'una dall'altra (data l'etichetta). In particolare, quasi tutti i problemi del mondo reale contengono funzioni con diversi gradi di dipendenza dagli altri. Se dovessimo evitare tutte le funzioni che erano dipendenti l'uno dall'altro, sarebbe stato molto difficile costruire insiemi di funzioni che forniscono le informazioni necessarie per l'apprendimento automatico dell'algoritmo.

Che cosa succede quando si ignora l'assunzione di indipendenza, e utilizziamo il classificatore Naive Bayes con funzioni che non sono indipendenti? Un problema che si pone è che il classificatore può far terminare in "doppio conteggio" l'effetto di funzioni altamente correlate, spingendo il classificatore più vicino ad una determinata etichetta.

Per vedere come questo si può verificare, si consideri un classificatore di genere di nome che contiene due funzioni identiche, f_1 e f_2 . In altre parole, f_2 è una copia esatta di f_1 , e non contiene nuove informazioni. Quando il classificatore prende in considerazione un input, includerebbe il contributo sia di f_1 e f_2 al momento di decidere quale etichetta scegliere. Pertanto, al contenuto informativo di queste due funzioni sarà dato più peso di quanto meriti.

Naturalmente, di solito non costruiamo classificatori naive Bayes che contengono due funzioni identiche. Tuttavia, facciamo costruire classificatori che contengono caratteristiche che sono dipendenti l'uno dall'altro. Per esempio, le caratteristiche che terminano-con (a) e terminano con-(vocale) sono interdipendenti, perché se un valore di imput ha la prima funzione, allora deve anche avere una seconda funzione. Per questo tipo di caratteristiche, le informazioni duplicate possono dare maggiore peso di quanto è stato giustificato dal set d lavoro.

Il problema del doppio conteggio

Il problema del doppio conteggio è che durante il lavoro, i contributi funzionali vengono calcolati separatamente; ma quando si usa il classificatore per scegliere le etichette per i nuovi input, tali contributi funzionali sono combinati. Una soluzione, quindi, è quello di considerare le possibili interazioni tra i contributi funzionali durante il lavoro. Potremmo quindi utilizzare queste interazioni per regolare i contributi che fanno le singole funzioni.

Per essere più precisi, è possibile riscrivere l'equazione utilizzata per calcolare la probabilità di una etichetta, separando il contributo di ciascuna funzione (o etichetta):

(7) $P\ (caratteristiche, etichetta) = w\ [etichetta] \times \text{Prod}_{f \mid a \mid \text{Caratteristiche}}\ w\ [f, etichetta]$

Qui, *w [etichetta]* è il "punteggio di partenza" per un dato marchio, e *w [f, etichetta]* è il contributo di una data caratteristica alla probabilità di un'etichetta. Noi chiamiamo questi valori *w [etichetta]* e *w [f, etichetta]* i **parametri** o **pesi** per il modello. Utilizzando l'algoritmo naive Bayes, abbiamo impostato ciascuno di questi parametri in modo indipendente:

(8) $w\ [label] = P\ (etichetta)$

(9) $w\ [f, label] = P\ (f \mid etichetta)$

Tuttavia, nel prossimo paragrafo, vedremo un classificatore che tenga conto delle possibili interazioni tra questi parametri quando scegliamo i loro valori.

6.6 Classificatori di massima entropia

Il classificatore di **massima entropia** utilizza un modello che è molto simile al modello impiegato dal classificatore naive Bayes. Ma invece di usare le probabilità per impostare i parametri del modello, utilizza tecniche di ricerca per trovare una serie di parametri in grado di ottimizzare le prestazioni del classificatore. In particolare, esso cerca l'insieme di parametri che ottimizza la **probabilità totale** del corpo di formazione, che è definito come:

(10) $P\ (caratteristiche) = \Sigma_{x \mid a \mid corpo}\ P\ (etichetta\ (x) \mid caratteristiche\ (x))$

Dove P (etichette | caratteristiche) , è la probabilità che un input le cui caratteristiche sono *caratteristiche*, l'etichetta di classe è *etichetta* , è definito come:

(11) $P\ (etichette \mid Voti) = P\ (etichette, caratteristiche) / \Sigma_{etichetta}\ P\ (etichette, caratteristiche)$

A causa delle interazioni potenzialmente complesse tra gli effetti di funzioni correlate, non c'è modo di calcolare direttamente i parametri del modello che ottimizzano la probabilità del set di lavoro. Pertanto, i classificatori di massima entropia scelgono i parametri del modello, utilizzando le tecniche di **massima iterazione** che inizializzano i parametri del modello di valori casuali, e poi ripetutamente affinano i parametri per avvicinarli alla soluzione ottimale. Queste tecniche di ottimizzazione iterativa garantiscono che ogni affinamento dei parametri si avvicinerà ai valori ottimali, ma non necessariamente forniscono un mezzo per determinare quando tali valori ottimali sono stati raggiunti. Poiché i parametri per classificatori di massima entropia sono stati selezionati usando tecniche iterative di ottimizzazione, possono richiedere molto tempo per imparare a usarle. Ciò è particolarmente vero quando la dimensione del set di lavoro, il numero di funzioni, e il numero di etichette, sono tutte grandi.

Nota

Alcune tecniche di ottimizzazione iterative sono molto più veloci di altre. Quando formiamo i modelli di massima entropia, evitiamo l'uso della Scala Iterativa Generalizzata(GIS) o della scala iterativa migliorata (IIS), che sono entrambi molto più lente del congiunto Gradiente (CG) e dei metodi di ottimizzazione BFGS.

Il modello di massima entropia

Il modello di classificazione di massima entropia è una generalizzazione del modello utilizzato dal classificatore Naive Bayes. Come il modello Naive Bayes, il classificatore massima entropia calcola la probabilità di ciascuna etichetta per un dato valore di input, moltiplicando insieme i parametri che sono applicabili per il valore di input e l'etichetta. Il modello del classificatore Naive Bayes definisce un parametro per ogni etichetta, specificando la probabilità a priori, e di un parametro per ogni (funzione, etichetta) coppia, specificando il contributo delle caratteristiche individuali nei confronti di probabilità di un'etichetta.

Al contrario, il modello di classificatore di massima entropia lascia all'utente di decidere quali combinazioni di etichette e caratteristiche vorrebbero ricevere i propri parametri. In particolare, è possibile utilizzare un singolo parametro per associare una funzione con più di una sola etichetta, o per associare più di una sola funzione con una determinata etichetta. Questo a volte permetterà al modello di "generalizzare" su alcune delle differenze tra le singole etichette o di funzioni.

Ogni combinazione di etichette e caratteristiche che riceve il proprio parametro è chiamata **funzione-comune**. Si noti che le funzioni comuni sono proprietà dei valori *etichettati*, mentre funzioni (semplici) sono proprietà di valori *non marcati*.

In genere, le funzioni comuni che vengono utilizzati per la costruzione di modelli di massima entropia rispecchiano esattamente quelli che vengono utilizzati dal modello Naive Bayes. In particolare, una funzione comune è definita per ogni etichetta, corrispondente a *w* [*etichetta*], e per ogni combinazione di (semplice) funzione e l'etichetta, corrispondente a *w* [*f* , *etichetta*]. Tenendo conto delle funzioni comuni per un modello di massima entropia, il punteggio assegnato a un'etichetta per un dato input è semplicemente il prodotto dei parametri associati alle funzioni comuni che si applicano a questo input e etichetta:

(12) *P (ingresso, etichetta)* = Prod. $_{\text{funzione-comune (ingresso, etichetta)}}$ *w [funzione-comune]*

Entropia ottimizzata

L'intuizione che motiva la classificazione di massima entropia è che dovremmo costruire un modello che cattura le frequenze delle singole caratteristiche comuni, senza fare nessuna ipotesi. Un esempio aiuterà a illustrare questo principio.

Supponiamo che vi viene assegnato il compito di raccogliere il senso corretto di una determinata parola, da un elenco di dieci sensi possibili (etichetta A-J). In un primo momento, non ci viene detto nulla di più sulla parola o dei sensi. Ci sono molte distribuzioni di probabilità che potremmo scegliere per i dieci sensi, come ad esempio:

Tabella 6.1

	A	B	C	D	E	F	G	H	I	J
(I)	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%
(Ii)	5%	15%	0%	30%	0%	8%	12%	0%	6%	24%
(Iii)	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%

Anche se qualcuna di queste distribuzioni *potrebbe* essere corretta, probabilmente scegliamo la distribuzione *(i)* , in quanto in assenza di più informazioni, non vi è motivo di ritenere che il senso di una parola è più probabile di altri. D'altra parte, distribuzioni *(ii)* e *(iii)* riflettono presupposti che non sono supportati da ciò che sappiamo.

Un modo per catturare l' intuizione che la distribuzione *(i)* è più "equa" rispetto agli altre due, è quello di richiamare il concetto di entropia. Nella discussione di alberi di decisione, abbiamo descritto l'entropia come una misura di quanto è stato "disorganizzato" un insieme di etichette. In particolare, se domina una sola etichetta, l'entropia è bassa, ma se le etichette sono più uniformemente distribuite, l'entropia è alta. Nel nostro esempio, abbiamo scelto la distribuzione *(i)*perché le sue probabilità di etichette sono equamente distribuite - in altre parole, perché la sua entropia è alta. In generale, il **principio d**i **massima entropia** afferma che, tra le distribuzioni che sono coerenti con quello che sappiamo, dobbiamo scegliere la distribuzione la cui entropia è massima.

Quindi, supponiamo che ci viene detto che il senso A appare il 55%. Ancora una volta, ci sono molte distribuzioni che sono coerenti con questo nuovo pezzo di informazioni, come ad esempio:

Tabella 6.2

	A	B	C	D	E	F	G	H	I	J
(Iv)	55%	45%	0%	0%	0%	0%	0%	0%	0%	0%
(V)	55%	5%	5%	5%	5%	5%	5%	5%	5%	5%
(Vi)	55%	3%	1%	2%	9%	5%	0%	25%	0%	0%

Ma ancora una volta, ci sarà probabilmente da scegliere la distribuzione che fa il minor numero di ipotesi infondate - in questo caso, di distribuzione *(v)* . Infine, supponiamo che ci viene detto che la parola "'up" appare nel contesto il 10% del tempo, e che, quando è

presente nel contesto c'è un 80% di possibilità che il senso A o C verranno utilizzati. Tuttavia, si può verificare la seguente distribuzione, che sembra più appropriata:

Tabella 6.3

		A	B	C	D	E	F	G	H	I	J
(Vii)	+ Up	5,1%	0,25%	2,9%	0,25%	0,25%	0,25%	0,25%	0,25%	0,25%	0,25%
``	- Up	49,9%	4,46%	4,46%	4,46%	4,46%	4,46%	4,46%	4,46%	4,46%	4,46%

In particolare, la distribuzione è coerente con quello che sappiamo: se sommiamo le probabilità nella colonna A, si ottiene il 55%, se sommiamo le probabilità della prima riga, si ottiene il 10%, e se sommiamo le caselle A e C nella riga più in alto, si ottiene l'8%. Inoltre, le probabilità rimanenti sembrano essere "distribuiti uniformemente."

In questo esempio, ci siamo limitati a distribuzioni che siano coerenti con quello che sappiamo, tra questi, abbiamo scelto la distribuzione con la massima entropia. Questo è esattamente ciò che il classificatore di massima entropia fa. In particolare, per ciascun caratteristica-comune, il modello di massima entropia calcola la "frequenza empirica" di tale elemento - cioè, la frequenza che si verifica nel set di lavoro. Si cerca quindi la distribuzione che massimizza l'entropia, pur prevedendo la frequenza corretta per ogni caratteristica comune.

Classificatori Generativi e quelli condizionali

Una differenza importante tra i classificatore Naive Bayes e il classificatore di massima entropia riguarda il tipo di domande che possono essere utilizzati per rispondere. Il classificatore naive Bayes è un esempio di un classificatore **generativo**, che costruisce un modello che predige *P (ingresso, etichetta)* , la comune probabilità di coppia *(input, etichetta)*. Di conseguenza, i modelli generativi possono essere utilizzato per rispondere alle seguenti domande:

1. Qual è l'etichetta più probabile per un dato input?
2. Quanto è probabile una determinata etichetta per un dato input?
3. Qual è il valore di input più probabile?
4. Come è probabile un dato valore di input?
5. Quanto è probabile che un dato valore di input con una determinata etichetta?
6. Qual è l'etichetta più probabile per un input che potrebbe avere uno dei due valori (ma non si sa quale)?

Il classificatore di massima entropia, invece, è un esempio di un **classificatore condizionale**. I Classificatori condizionali costruiscono modelli che predicono *P (etichetta* | *ingresso)* - la probabilità di un marchio di un dato valore di input. Così, i modelli condizionali possono essere utilizzati per rispondere alle domande 1 e 2.Tuttavia, i modelli condizionali possono *non* essere utilizzate per rispondere alle domande restanti 3-6.

In generale, i modelli generativi sono strettamente più potenti rispetto ai modelli condizionali, dal momento che siamo in grado di calcolare la probabilità condizionale *P (etichetta* | *ingresso)* dalla probabilità congiunta *P (ingresso, etichetta)* , ma non viceversa. Tuttavia, questa potenza aggiuntiva ha un prezzo. Poiché il modello è più

potente, ha più "parametri liberi", che devono essere appresi. Tuttavia, la dimensione del set di lavoro è fisso. Come risultato, un modello generativo non può fare un buon lavoro a rispondere alle domande 1 e 2 come un modello condizionale, in quanto il modello condizionale può concentrare i propri sforzi su queste due questioni. Tuttavia, se abbiamo bisogno di risposte a domande come 3-6, dobbiamo utilizzare un modello generativo.

La differenza tra un modello generativo e un modello condizionale è analoga alla differenza tra una carta topografica e una foto di un orizzonte. Sebbene la mappa topografica può essere utilizzato per rispondere a una più ampia varietà di questioni, è molto più difficile da generare una mappa accurata topografica che è di generare un orizzonte preciso.

6.7 Esempi di modellazione linguistica

I Classificatori possono aiutarci a comprendere i modelli linguistici che si verificano nel linguaggio naturale, consentendo di creare espliciti **modelli** che catturano quei modelli. In genere, questi esempi. Tipicamente, questi modelli sono usati per le tecniche di classificazione supervisionata, ma è anche possibile costruire analiticamente modelli motivati. In entrambi i casi, questi modelli espliciti servono a due scopi importanti: ci aiutano a capire i modelli linguistici, e possono essere usate per fare previsioni su nuovi dati linguistici.

La misura che i modelli intuiti possono darci in schemi linguistici, dipende in gran parte dal tipo di modello che viene utilizzato. Alcuni modelli, come gli alberi decisionali, sono relativamente trasparenti, e ci danno informazioni dirette su quali fattori sono importanti nel prendere decisioni e su quali fattori sono legati l'uno all'altro. Altri modelli, come le reti neurali multi-livello, sono molto più opachi. Anche se può essere possibile ottenere informazioni dal loro studio, che in genere prende molto più lavoro.

Ma tutti gli espliciti modelli possono fare previsioni su nuovi " **invisibili** " dati linguistici che non erano stati inclusi nel corpo utilizzato per creare il modello. Queste previsioni possono essere valutate per stimare l'accuratezza del modello. Una volta che un modello si ritiene sufficientemente accurato, può quindi essere utilizzato per prevedere automaticamente le informazioni su nuovi dati linguistici. Questi modelli predittivi possono essere combinati in sistemi che eseguono molte utili funzioni di elaborazione del linguaggio, come la classificazione dei documenti, la traduzione automatica, e domande corrispondenti.

Che cosa ci dicono i modelli?

E 'importante capire che cosa possiamo imparare sulla lingua da un modello costruito automaticamente. Una considerazione importante quando si tratta di modelli di linguaggio è la distinzione tra modelli descrittivi e modelli esplicativi. I modelli descrittivi catturano i modelli nei dati, ma non fornisce alcuna informazione sul *motivo per cui* i dati contengono quegli schemi. Per esempio, come abbiamo visto nella tabella 3.1 , i sinonimi *assolutamente* e *sicuramente* non sono intercambiabili: si dice *assolutamente adoro* non *sicuramente adorano* , e *sicuramente preferisco* non *assolutamente preferisce* . Al contrario, i modelli esplicativi tentano di catturare le proprietà e le relazioni che causano i modelli linguistici. Per esempio, si potrebbe introdurre il concetto astratto di "aggettivo polare", come uno che ha un significato estremo, e classificare alcuni aggettivi come *adoro* e *detesto* come polare. Il nostro modello esplicativo dovrebbe contenere il vincolo che *assolutamente* non può essere combinato agli aggettivi polari,

e *sicuramente* non può essere combinato con aggettivi non polari. In sintesi, i modelli descrittivi forniscono informazioni sulle correlazioni tra i dati, mentre i modelli esplicativi vanno oltre per postulare le relazioni causali.

La maggior parte dei modelli che vengono costruiti da un corpo sono modelli descrittivi, in altre parole, possono dirci quali caratteristiche sono rilevanti per un dato modello o per una data costruzione, ma non possono necessariamente dirci come quelle caratteristiche e modelli si relazionano tra loro. Se il nostro obiettivo è quello di comprendere i modelli linguistici, allora possiamo utilizzare queste informazioni su quali sono correlate funzioni come punto di partenza per ulteriori esperimenti, volti a prendere in giro le relazioni tra caratteristiche e modelli. D'altra parte, se siamo solo interessati a utilizzare il modello per fare previsioni (ad esempio, come parte di un sistema di elaborazione del linguaggio), allora si può utilizzare il modello per fare previsioni su nuovi dati senza preoccuparsi dei dettagli delle casuali relazioni.

6.8 Sommario

- Modellazione dei dati linguistici che si trovano nel corpo può aiutarci a capire i modelli linguistici, e può essere usato per fare previsioni su nuovi dati linguistici.
- Classificatori supervisionati utilizzano i corpi di formazione dell'etichettata per costruire modelli che predicono l'etichetta di un input in base a caratteristiche specifiche di tale input.
- Classificatori supervisionati sono in grado di eseguire una vasta gamma di compiti di PNL, compresa la classificazione dei documenti, parte del discorso aggiunto, segmentazione frase, atto di identificazione del tipo di dialogo, le relazioni determinanti, e molti altri compiti.
- Quando formiamo un classificatore supervisionato, si dovrebbe dividere il corpo in tre set di dati: un set di lavoro per la costruzione del modello di classificatore, un insieme di dev- test per aiutare a selezionare e mettere a punto le caratteristiche del modello, e un insieme di test per valutare le prestazioni del modello finale.
- Quando si valuta un classificatore supervisionato, è importante utilizzare dati aggiornati, che non sono stati inclusi nel set di lavoro o dev-test. In caso contrario, i risultati della valutazione possono essere irrealisticamente ottimisti.
- Alberi di decisione vengono costruite con diagrammi di flusso con struttura ad albero che vengono utilizzati per assegnare etichette ai valori di input in base alle loro caratteristiche. Anche se sono di facile interpretazione, non sono molto bravi a gestire i casi in cui i valori delle funzioni interagiscono nel determinare l'etichetta corretta.
- Nei classificatori naive Bayes, ogni caratteristica contribuisce in modo indipendente per decidere quale etichetta deve essere utilizzata. Questo permette alle funzioni e ai valori di interagire, ma può essere problematico quando due o più funzioni sono altamente correlati tra loro.
- Classificatori di Massima Entropia utilizzano un modello di base che è simile al modello utilizzato da naive Bayes, tuttavia, impiegano un ottimizzazione iterativa per trovare l'insieme di caratteristiche che ottimizzano la probabilità del set di lavoro.
- La maggior parte dei modelli che vengono costruiti da un corpo sono descrittivi - ci hanno fatto sapere quali caratteristiche sono rilevanti per un dato modello o una data costruzione, ma non danno alcuna informazione sulle relazioni causali tra tali caratteristiche e modelli.

6.9 Approfondimenti

Si prega di consultare http://www.nltk.org/ per ulteriori informazioni su questo capitolo e su come installare i pacchetti esterni di apprendimento automatico, come Weka, Mallet, TADM e MEGAM. Per ulteriori esempi di classificazione e di apprendimento automatico con NLTK, consultare gli HOWTO alla pagina http://www.nltk.org/howto .

Per una introduzione generale all'apprendimento automatico, si consiglia (Alpaydin, 2004) . Per un'introduzione più matematicamente intenso alla teoria dell'apprendimento, vedere (Hastie, Tibshirani, e Friedman, 2009) . Eccellenti libri per utilizzare tecniche di apprendimento per NLP sono (Abney, 2008) , (Daelemans & Bosch, 2005) , (Feldman e Sanger, 2007) , (Segaran, 2007) , (Weiss et al, 2004) . Per ulteriori informazioni su tecniche di regolarità per problemi di lingua, vedere (Manning & Schutze, 1999) . Per ulteriori informazioni sui modelli di sequenza, e in particolare modelli di Markov, vedere(Manning & Schutze, 1999) o (Jurafsky & Martin, 2008) . il Capitolo 13 (Manning, Raghavan, e Schutze, 2008) discute l'uso di Naive Bayes per la classificazione dei testi.

Molto sull'apprendimento degli algoritmi trattati in questo capitolo sono numericamente intensivi, e, di conseguenza, verrà eseguito lentamente quando verrà codificato in Python. Per informazioni su come aumentare l'efficienza degli algoritmi numericamente intensivi in Python, vedere (Kiusalaas, 2005) .

Le tecniche di classificazione descritte in questo capitolo possono essere applicato ad una grande varietà di problemi. Per esempio, (Agirre & Edmonds, 2007) utilizza classificatori per eseguire le disambiguazioni del senso della parola e (Melamed, 2001) utilizza classificatori per creare testi paralleli. Libri di testo più recenti che parlano della classificazione del testo sono (Manning, Raghavan, e Schutze, 2008) e (Croft, Metzler, e Strohman, 2009) .

Gran parte dell' attuale ricerca nell'applicazione di tecniche di apprendimneto per i problemi di PNL è guidata da competizioni "sfide", in cui un insieme di organizzazioni di ricerca sono tutti forniti con il corpo stesso di sviluppo, e ha chiesto di costruire un sistema, e il conseguente sistema viene confrontato sulla base di una serie di test riservati. Esempi di queste sfide di competizioni CoNLL includono attività condivise, i concorsi, le gare ACE Riconoscendo implicazione testuali, e le competizioni AQUAINT.

Consultare http://www.nltk.org/

6.10 Esercizi

1. ☼ Leggi su una delle tecnologie linguistiche menzionate in questa sezione, come la disambiguazione del senso della parola, il ruolo semantico dell'etichettatura, domande corrispondenti, traduzione automatica, rilevazione dell'entità denominata. Scopri che tipo e la quantità di dati annotati è necessaria per lo sviluppo di tali sistemi. Perché pensi che una grande quantità di dati sia necessario?
2. ☼ L'utilizzo di qualsiasi dei tre classificatori descritti in questo capitolo, e tutte le funzioni a cui si può pensare, si può costruire meglio il classificatore genere del nome. Inizia suddividendo il Corpo Nomi in tre sottoinsiemi: 500 parole per il set di prova, 500 parole per l'insieme dev-test, e le restanti 6.900 parole per il set di

lavoro. Poi,ad esempio partendo con il classificatore di genere nome, apportare miglioramenti incrementali. Utilizzare l'insieme del dev-test per controllare i vostri progressi. Quando si è soddisfatti con il vostro classificatore, controllarne lo svolgimento finale sull'insieme di test. Come sono le prestazioni sul set di prova? confrontare la performance dell'insieme dev-test. E 'questo quello che ci si aspetta?

3. ☼ il corpo, La SENSEVAL 2,contiene dati destinati a formare il classificatore della disambiguazione del senso della. Esso contiene i dati di quattro parole: duro, interesse, linea, e servo. Scegli una di queste quattro parole, e carica i dati corrispondenti:

```
>>> da nltk.corpus import SENSEVAL
 >>> istances = senseval.instances ( 'hard.pos' )
 >>> size = int (len (istanze) * 0,1)
 >>> train_set, test_set = istances [dimension:] , le istanze [:
dimensions]
```

Con questo insieme di dati, costruire un classificatore che prevede il senso corretto del tag per una determinata istanza. Leggere la Guida al corpushttp://www.nltk.org/howto

4. ☼ usando la revisione del film discusso in questo capitolo, generare un elenco delle 30 caratteristiche che il classificatore trova più importanti. Puoi spiegare perché queste particolari caratteristiche sono importanti?
5. ☼ Selezionare uno dei compiti di classificazione descritte in questo capitolo, come il rilevamento nome genere, classificazione dei documenti, parte delle parole aggiunte, o atto di classificazione di dialogo. Con la stessa formazione e dati di prova, e le stesse caratteristiche di estrazione, costruire tre classificatori per l'attività: un albero decisionale, un classificatore Naive Bayes, e un classificatore di massima entropia. Confrontare le prestazioni dei tre classificatori sulla vostra attività selezionata. Come si fa a pensare che i risultati potrebbero essere diversi se si è utilizzato una caratteristica di estrazione diversa?
6. ☼ Il sinonimi *forte* e *potente* modelli diversi (provare a combinare con *trucioli* e *vendita*). Quali caratteristiche sono rilevanti in questa distinzione? Costruire un classificatore che prevede, quando ogni parola deve essere usata.
7. ◑ Il classificatore dell'atto di dialogo assegna delle etichette alle singole pagine, senza considerare il contesto in cui si trova. Tuttavia gli atti di dialogo sono fortemente dipendenti dal contesto, e alcune sequenze di atti di dialogo sono molto più probabili di altre. Approfittate di questo fatto per costruire un classificatore per l'etichettatura di atti di dialogo. Assicurati di considerare quali caratteristiche potrebbe essere utili. Vedere il codice per il classificatore consecutivo nella parte della parola aggiunte nell' Esempio 6.7 per ottenere alcune idee.
8. Funzionalità di Word ◑ può essere molto utile per l'esecuzione di classificazione dei documenti, in quanto le parole che compaiono in un documento possono dare un forte segnale a quello che è il suo contenuto semantico. Tuttavia, molte parole si verificano molto raramente, e alcune delle parole più importanti in un documento non può essere verificato nei nostri dati di lavoro. Una soluzione è quella di fare uso di un **lessico** , che descrive come diverse parole sono in relazione tra loro. Con lessico WordNet, possiamo aumentare la revisione di un film presentato in questo capitolo per utilizzare le funzioni che generalizzano le parole che compaiono in un documento, rendendo più probabile che essi corrispondono alle parole presenti nei dati di lavoro.

9. ★ Il corpo PP allegato, è un corpo che descrive preposizionali frasi di decisione. Ogni istanza nel corpo è codificato come un oggetto *PPallegato*:

```
>>> from nltk.corpus import ppattach
>>> ppattach.attachments('training')
[PPAttachment(sent='0', verb='join', noun1='board',
              prep='as', noun2='director', attachment='V'),
 PPAttachment(sent='1', verb='is', noun1='chairman',
              prep='of', noun2='N.V.', attachment='N'),
 ...]
>>> inst = ppattach.attachments('training')[1]
>>> (inst.noun1, inst.prep, inst.noun2)
('chairman', 'of', 'N.V.')
```

Selezionare solo i casi in cui *inst.allegato* è n :

```
>>> nattach = [inst for inst in ppattach.attachments('training')
...             if inst.attachment == 'N']
```

L'utilizzo di questo sub-corpo, costruisce un classificatore che tenta di prevedere quale preposizione viene utilizzata per collegare una determinata coppia di sostantivi. Per esempio, data la coppia di sostantivi "squadra" e "ricercatori", il classificatore deve prevedere la preposizione "di". Leggere la Guida al corpushttp://www.nltk.org/howto

10. ★ Si supponga di voler generare automaticamente una descrizione in prosa di una scena, e già si ha una parola per descrivere in modo univoco ogni entità, come ad esempio *il vaso* , e voleva semplicemente decidere se utilizzare *in* o *in* relazione a varie voci, ad esempio, *il libro è nella credenza* contro *il libro è sullo scaffale* . Esplora questo tema, cercando nel corpo di dati; la scrittura di programmi in base alle esigenze:
    a. In macchina rispetto sul treno
    b. In città rispetto in campagna
    c. Nella figura rispetto sullo schermo
    d. Nel Macbeth rispetto il Letterman

Circa questo documento ...

Questo documento è Revisione: 8464 lun 14 dicembre 2009 10:58:42 CET

Capitolo 7

Capitolo 8

Capitolo 9

Capitolo 10

Analisi del significato delle frasi

Abbiamo visto quanto sia utile sfruttare la potenza di un computer per elaborare il testo su larga scala. Tuttavia, ora che conosciamo i meccanismi di analisi sintattica (parser) e le feature based grammars, possiamo fare qualcosa di ugualmente utile analizzando il significato delle frasi? L'obiettivo di questo capitolo è rispondere alle seguenti domande:

1. Come possiamo rappresentare il significato del linguaggio naturale in modo tale che un computer sia in grado di elaborare queste rappresentazioni?

2. Come possiamo associare rappresentazioni del significato con una serie illimitata di frasi?

3. Come possiamo usare programmi che collegano le rappresentazioni del significato delle frasi ad una conoscenza pregressa?

Lungo il percorso impareremo alcune tecniche formali nel campo della logica semantica e vedremo come queste possano essere utilizzate per interrogare i database che memorizzano i fatti riguardanti il mondo.

10.1 Comprensione del linguaggio naturale

Interrogare un Database

Supponiamo di avere un programma che ci permetta di digitare una domanda utilizzando il linguaggio naturale e che ci fornisca la risposta giusta:

(1)

a. Which country is Athens in? (In quale nazione si trova Atene?)

b. Greece. (In Grecia)

Quanto è difficile scrivere un programma come questo? E ci può bastare usare le stesse tecniche che abbiamo incontrato finora in questo libro, o si necessita qualcosa di nuovo? In questa sezione, si mostrerà che risolvere il compito in un dominio ristretto è piuttosto semplice. Ma vedremo anche che per affrontare il problema in modo più generale, dobbiamo aprire una nuova scatola di idee e tecniche, che coinvolgono la rappresentazione del significato.

Iniziamo supponendo di avere i dati relativi alle città e ai paesi in una forma strutturata. Per essere concreti, utilizzeremo una tabella di database di cui riportiamo le prime righe sono nella tabella 10.1.

Nota

I dati illustrati nella tabella 10.1 sono tratti dal sistema di Chat-80 (Warren & Pereira, 1982). Il numero della popolazione è espresso in migliaia, ma si noti che i dati utilizzati in questi esempi risalgono almeno agli anni ottanta e che erano già un po' obsoleti nel periodo in cui (Warren & Pereira, 1982) è stato pubblicato.

Tabella 10.1:

city_table: una tabella delle città, paesi e popolazioni

City	**Country**	**Population**
athens	greece	1368
bangkok	thailand	1178
barcelona	spain	1280
berlin	east_germany	3481
birmingham	united_kingdom	1112

Il modo più ovvio per recuperare le risposte da questi dati tabulari prevede la scrittura di domande in un linguaggio query di database, quale SQL.

Nota

SQL (Structured Query Language) è un linguaggio progettato per il recupero e la gestione dei dati nei database relazionali. Se volete saperne di più su SQL, il sito http://www.w3schools.com/sql/ è un buon punto di riferimento.

Ad esempio, eseguendo la query (2) si otterrà il valore `'greece'` :

(2)
```
SELECT Country FROM city_table WHERE City = 'athens'
```

Questo consente di specificare un set di risultati composto da tutti i valori per la colonna paese in righe di dati in cui il valore della colonna città è 'Atene'.

Come si può ottenere lo stesso effetto usando l'inglese come nostro input per interrogare il sistema? Il formalismo della feature-based grammar rende semplice la traduzione dall'inglese a SQL. La grammatica sql0.fcfg illustra come mettere insieme la rappresentazione del significato di una frase con l'analisi della frase. Ogni regola di struttura della frase è completata da un'indicazione per la costruzione del valore per la caratteristica sem. Le indicazioni sono estremamente semplici;in ogni caso si utilizza il simbolo + per sommare i valori che costituiscono il figlio e ottenere così un valore che costituisce il genitore.

```
>>> nltk.data.show_cfg('grammars/book_grammars/sql0.fcfg')
% start S
S[SEM=(?np + WHERE + ?vp)] -> NP[SEM=?np] VP[SEM=?vp]
VP[SEM=(?v + ?pp)] -> IV[SEM=?v] PP[SEM=?pp]
VP[SEM=(?v + ?ap)] -> IV[SEM=?v] AP[SEM=?ap]
NP[SEM=(?det + ?n)] -> Det[SEM=?det] N[SEM=?n]
PP[SEM=(?p + ?np)] -> P[SEM=?p] NP[SEM=?np]
AP[SEM=?pp] -> A[SEM=?a] PP[SEM=?pp]
NP[SEM='Country="greece"'] -> 'Greece'
```

```
NP[SEM='Country="china"'] -> 'China'

Det[SEM='SELECT'] -> 'Which' | 'What'

N[SEM='City FROM city_table'] -> 'cities'

IV[SEM=''] -> 'are'

A[SEM=''] -> 'located'

P[SEM=''] -> 'in'
```

Questo ci permette di analizzare una query in SQL.

```
>>> from nltk import load_parser

>>> cp = load_parser('grammars/book_grammars/sql0.fcfg')

>>> query = 'What cities are located in China'

>>> trees = cp.nbest_parse(query.split())

>>> answer = trees[0].node['SEM']

>>> q = ' '.join(answer)

>>> print q

SELECT City FROM city_table WHERE Country="china"
```

Nota

Il tuo turno: esegui il parser con la massima tracciatura, vale a dire **cp = load_parser ('grammars/book_grammars/sql0.fcfg', trace = 3)** ed esamina come i valori di sem, costruiti come estremi completi, vengono aggiunti al grafico.

Infine, eseguiamo la query sul database city.db e recuperiamo alcuni risultati.

```
>>> from nltk.sem import chat80

>>> rows = chat80.sql_query('corpora/city_database/city.db', q)

>>> for r in rows: print r[0],
```

canton chungking dairen harbin kowloon mukden peking shanghai sian tientsin

Since each row r is a one-element tuple, we print out the member of the tuple rather than tuple itself

Per riassumere, abbiamo definito la funzione per cui un computer restituisce dati utili in risposta a una query in linguaggio naturale, e l'abbiamo implementata traducendo un piccolo sottoinsieme di inglese in SQL. Possiamo dire che il nostro codice NLTK già "capisce" SQL, dato che Python è in grado di eseguire la query SQL in un database, e per estensione "capisce" anche domande come *What cities are located in China? (Quali città si trovano in Cina?)*. This parallels being able to translate from Dutch into English as an example of natural language understanding. Supponiamo che tu sia un madrelingua inglese e che tu abbia cominciato a imparare l'olandese. L'insegnante ti domanda se capisci cosa (3) significa:

(3) Margrietje houdt van Brunoke.

Se conosci il significato delle singole parole (3)e sai come questi elementi sono combinati per rendere il significato della frase intera, potresti dire che (3) ha lo stesso senso di

Margrietje loves Brunoke.

Un osservatore — che chiameremo Olga — potrebbe anche considerare questo come la prova che tu hai afferrato il significato di (3), ma tutto ciò dipende comunque dalla stessa comprensione dell'inglese da parte di Olga. Se Olga non ti comprende, la tua traduzione dall'olandese all'inglese non la convincerà circa la tua capacità di capire l'olandese. Torneremo su questo punto tra poco.

La grammatica sql0.fcfg, insieme al parser *NLTK Earley*, è fondamentale per effettuare la traduzione dall'inglese in SQL. Quanto è adeguata questa grammatica? Si è visto che la traduzione di SQL per l'intera frase è stata costruita dalla traduzione dei componenti. Tuttavia, sembrerebbe che non ci siano tante giustificazioni per queste rappresentazioni di significato dell'elemento. Ad esempio, se guardiamo l'analisi del sintagma *What cities*, il determinante e il sostantivo corrispondono rispettivamente ai frammenti di SQL *Select* e *from city_table*. Niente di tutto ciò, però, ha un significato ben definito se isolato dagli altri elementi.

Si può avanzare anche un'altra critica alla grammatica: abbiamo "cablato" un'imbarazzante quantità di dettagli riguardo a quello che è all'interno del database. Abbiamo bisogno di conoscere il nome della tabella pertinente (ad esempio, city_table) e i nomi dei campi. Il nostro database, però, potrebbe contenere le stesse righe di dati usando solo un nome differente per la tabella e per il campo e, in questo caso, SQL query non sarebbe eseguibile. Allo stesso modo, potremmo conservare i nostri dati in un formato diverso, ad esempio XML, e in questo caso recuperare gli stessi risultati richiederebbe di tradurre le nostre domande dall'inglese in un linguaggio di query XML anziché SQL. Queste

considerazioni suggeriscono che noi dovremmo tradurre l'inglese in qualcosa che è più astratto e generico di SQL.

Allo scopo di centrare il punto, prendiamo in considerazione un'altra query inglese e la sua traduzione:

(4)

a. What cities are in China and have population above 1.000.000?

(Quali città sono in Cina e hanno una popolazione superiore al 1.000.000?)

b.
```
SELECT City FROM city_table WHERE Country = 'china' AND Population > 1000
```

Nota

Il tuo turno: estendi la grammatica sql0.fcfg in modo che (4a) si traduca in (4b) e controlla i valori restituiti dalla query.

Probabilmente troverete più facile estendere in primo luogo la grammatica per gestire le query come *Quali città hanno una popolazione superiore al 1.000.000* prima di affrontare la congiunzione. Dopo aver fatto il tentativo, è possibile confrontare la soluzione su grammars/book_grammars/sql1.fcfg nella distribuzione dei dati NLTK.

Osserva che la congiunzione *and* nella (4a) è tradotta in AND dalla controparte SQL, (4b). Quest'ultimo ci dice di selezionare i risultati dalle righe dove due condizioni sono vere insieme: il valore della colonna paese è 'Cina' e il valore della colonna popolazione è maggiore di 1000. Questa interpretazione di *and* mette in campo una nuova idea: ci parla di ciò che è vero in qualche situazione particolare, e ci dice che Cond1 AND Cond2 è vero nella situazione s solo nel caso in cui la condizione *Cond1* è vera in *s* e la condizione *Cond2* è vera in *s*. Anche se questo non tiene conto dell'intera gamma di significati di *and* in inglese, ha la buona proprietà di essere indipendente da qualsiasi linguaggio di query. Infatti, gli abbiamo dato l'interpretazione standard dalla logica classica. Nelle sezioni seguenti, esploreremo un approccio in cui vengono tradotte le frasi del linguaggio naturale in logico invece di un linguaggio di query eseguibile, come ad esempio SQL. Uno dei vantaggi dei formalismi logici è che sono più astratti e quindi più generici. Se volessimo, una volta ottenuta la nostra traduzione in linguaggio logico, potremmo quindi tradurla in diverse altre lingue speciali. Difatti, i più seri tentativi di interrogazione di database tramite il linguaggio naturale hanno utilizzato questa metodologia.

Linguaggio Naturale, Semantico e Logico

Siamo partiti cercando di catturare il significato di (1a) traducendolo in una richiesta in un altro linguaggio, SQL, che il computer può interpretare ed eseguire. Questo, però, ci lascia ancora con il dubbio che la traduzione fosse corretta. Facendo un passo indietro alla query al database, notiamo che il significato di *and* sembra dipendere dalla possibilità di essere in grado di specificare quando le affermazioni sono vere o meno in una particolare situazione. Invece di tradurre una frase *S* da un linguaggio ad un altro, cerchiamo di dire di cosa riguarda *S* mettendola in relazione con una situazione nel mondo. Portiamo ulteriormente avanti questo discorso. Immaginiamo una situazione *s* in cui ci sono due entità, Margrietje e la sua bambola preferita, Brunoke. Aggiungiamo che c'è un rapporto che unisce le due entità, che chiameremo relazione d'amore. Se comprendi il significato di (3) sai che questo è vero nella situazione *s*. In parte, già sei a conoscenza di questo perché sai che *Margrietje* si riferisce a Margrietje, *Brunoke* si riferisce a Brunoke, e *houdt van* si riferisce alla relazione d'amore.

Abbiamo introdotto due nozioni fondamentali di semantica. La prima è che le frasi dichiarative sono vere o false in determinate situazioni. La seconda è che definite frasi nominali e nomi propri si riferiscono a cose del mondo. Quindi la situazione (3) vera nel caso in cui Margrietje amasse la bambola Brunoke è illustrata nella Figura 10.1.

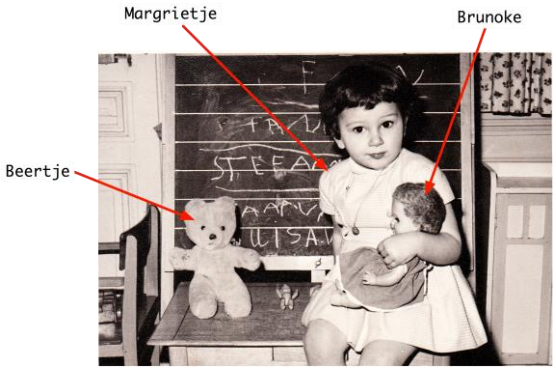


Figura 10.1: rappresentazione della situazione in cui Margrietje ama Brunoke.

Una volta adottata la nozione di verità in una situazione, abbiamo un potente strumento per ragionare. In particolare, possiamo guardare a insiemi di frasi e domandarci se potrebbero essere vere insieme in una qualsiasi situazione. Ad esempio, le frasi in (5) possono essere entrambe vere, mentre quelle in (6) e (7) non possono. In altre parole, le frasi in (5) sono **coerenti**, mentre quelle in (6) e (7) sono **incoerenti**.

(5) a. Sylvania is to the north of Freedonia. (Sylvania è a nord delle Freedonia.)

b. Freedonia is a republic. (Freedonia è una repubblica.)

(6) a. The capital of Freedonia has a population of 9.000. (La capitale della Freedonia ha una popolazione di 9.000 abitanti.)

b. No city in Freedonia has a population of 9.000. (Nessuna città in Freedonia ha una popolazione di 9.000 abitanti.)

(7) a. Sylvania is to the north of Freedonia. (Sylvania è a nord di Freedonia.)

b. Freedonia is to the north of Sylvania. (Freedonia è a nord di Sylvania.)

Abbiamo scelto frasi riguardanti paesi immaginari (presenti nel film del 1933 dei fratelli Marx, *Duck Soup*) per sottolineare che la capacità di ragione su questi esempi non dipende da ciò che è vero o falso nel mondo reale. Se si conosce il significato della parola *no* e si sa che la capitale di una nazione è una città che si trova in quella nazione, allora si dovrebbe essere in grado di concludere che le due frasi (6) sono incoerenti, indipendentemente da dove si trovi Freedonia o a che numero ammonti la popolazione della sua capitale. Ovvero, non c'è nessuna possibile situazione in cui entrambe le frasi potrebbero essere vere. Analogamente, se si comprende che la relazione di spazio espressa dalla frase *a nord di* è asimmetrica, si dovrebbe essere in grado di concludere che le due frasi (7) sono incompatibili.

Parlando genericamente, gli approcci al linguaggio naturale basati sulla logica si concentrano su quegli aspetti del linguaggio naturale che guidano il nostro giudizio riguardo la coerenza o l'incoerenza. La sintassi del linguaggio logico è progettata per esplicitare formalmente queste caratteristiche. As a result, determining properties like consistency can often be reduced to symbolic manipulation, that is, to a task that can be carried out by a computer. Allo scopo di perseguire questo approccio, vogliamo prima sviluppare una tecnica per la rappresentazione di una situazione possibile. Lo facciamo nei termini di ciò che gli studiosi di logica chiamano "modello".

Un **modello** per un insieme di frasi *w* è una rappresentazione formale di una situazione in cui tutte le frasi in *w* sono vere. Il modo solito di rappresentare i modelli coinvolge la teoria degli insiemi. Il dominio *D* del discorso (tutte le entità che ci interessano attualmente) è un insieme di individui, mentre le relazioni vengono trattate come insiemi derivanti da *D*. Vediamo un esempio concreto. Il nostro dominio *D* è composta da tre bambini, Stefan, Klaus ed Evi, rappresentati rispettivamente come s, k ed e. Possiamo schematizzare in questo modo *D* = {s, k, e}. L'espressione *boy* denota l'insieme composto da Stefan e Klaus, l'espressione *girl* indica l'insieme composto da Evi, e l'espressione *is running* indica l'insieme costituito da Stefan ed Evi. La figura 10.2 un rendering grafico del modello.

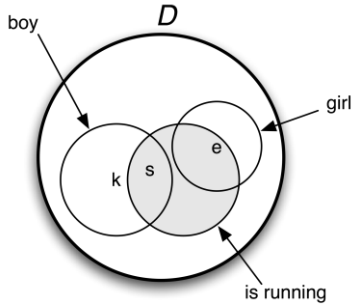


Figura 10.2: diagramma di un modello contenente un dominio *D* e i sottoinsiemi di *D* corrispondenti ai termini: *boy, girl, is running*

Più avanti nel capitolo useremo dei modelli per contribuire alla valutazione della veridicità o falsità delle frasi inglesi e, in questo modo, riusciremo ad illustrare alcuni dei metodi per la rappresentazione del significato. Tuttavia, prima di scendere nei dettagli, allarghiamo la prospettiva di discussione e ricolleghiamoci ad un argomento che è stato sommariamente trattato nella sezione 1.5. Un computer può capire il significato di una frase? E come possiamo dire che lo abbia fatto? Questo equivale a domandarsi "Un computer può pensare?". Com'è noto, Alan Turing propose di rispondere a questa domanda esaminando la capacità di un computer di intrattenere una conversazione sensata con un umano (Turing, 1950). Si supponga di sostenere due conversazioni, una con una persona e una con un computer, senza però sapere chi sia l'una e chi l'altro. Se dopo aver chiacchierato con entrambe non si riesce ad identificare quale dei partner sia la macchina, vorrà dire che il computer ha imitato con successo un essere umano. Nel caso in cui il computer riuscisse a farsi passare come un umano in questo "gioco d'imitazione" (noto come "Test di Turing"), secondo Turing dovremmo essere pronti a dire che il computer *può* pensare e può definirsi dunque intelligente. In tal maniera, Turing ha eluso qualsiasi questione di esame degli stati interni di un computer utilizzando invece il suo *comportamento* come prova di intelligenza. Partendo dallo stesso ragionamento, si desume che per dire che un computer capisce inglese, ha solo bisogno di comportarsi come se lo sapesse fare. Quello che qui importa non sono tanto le specifiche del gioco d'imitazione di Turing, quanto la proposta di giudicare la capacità di comprensione del linguaggio naturale nei termini di un comportamento osservabile.

10.2 La Logica Proposizionale

Un linguaggio logico è progettato per rendere il ragionamento formalmente esplicito. Di conseguenza, è in grado di catturare quegli aspetti del linguaggio naturale che determinano se un insieme di frasi è coerente. Come parte di questo approccio, abbiamo bisogno di sviluppare rappresentazioni logiche di una frase ϕ che formalmente raggruppi le **condizioni di verità** di ϕ. Cominceremo con un semplice esempio:

(8) [Klaus chased Evi] and [Evi ran away]

Sostituiamo le due frasi in (8) rispettivamente con ϕ e ψ e mettiamo & per indicare l'operatore logico corrispondente alla parola inglese *and*: ϕ & ψ. Questa struttura è la **forma logica** di (8).

La **logica proposizionale** permette di rappresentare solo quelle parti della struttura linguistica che corrispondono a determinati connettori sentenziali. Abbiamo appena visto *and*. Altri tali connettivi sono *not*, *or* e *if..., then...*. Nella formalizzazione della logica proposizionale, le controparti di tali connettori sono a volte chiamate **operatori booleani**. Le espressioni di base della logica proposizionale sono i **simboli proposizionali**, spesso scritti come *P*, *Q*, *R*, ecc. Ci sono vari modi convenzionali per la rappresentazione degli operatori booleani. Dal momento in cui ci concentreremo sui modi di esplorare la logica all'interno di NLTK, ci atterremo alle seguenti versioni ASCII degli operatori:

```
>>> nltk.boolean_ops()
negation            -
conjunction         &
disjunction         |
implication         ->
equivalence         <->
```

Dai simboli proposizionali e dagli operatori booleani, si può costruire un insieme infinito di FORMULE CORRETTE (o anche solo formule, per abbreviare) di logica proposizionale. Quindi se ϕ è una formula, allo stesso modo lo è ϕ -. E se ϕ e ψ sono formule, così lo sono (ϕ & ψ) (ϕ | ψ) (ϕ -> ψ) (ϕ <> - ψ).

La tabella 10.2 consente di specificare le condizioni di verità per le formule che contengono questi operatori. Come prima usiamo ϕ e ψ come variabili sulle frasi, e abbreviamo *if and only if* come *iff*.

Tabella 10.2:

Condizioni di verità per gli operatori booleani nella logica proposizionale.

Boolean Operator		**Truth Conditions**	
negation (*it is not the case that ...*)	-ϕ is true in *s*	iff	ϕ is false in *s*
conjunction (*and*)	(ϕ & ψ) is true in *s*	iff	ϕ is true in *s* and ψ is true in *s*
disjunction (*or*)	(ϕ | ψ) is true in *s*	iff	ϕ is true in *s* or ψ is true in *s*
implication (*if ..., then ...*)	(ϕ -> ψ) is true in *s*	iff	ϕ is false in *s* or ψ is true in *s*
equivalence (*if and only if*)	(ϕ <-> ψ) is true in *s*	iff	ϕ and ψ are both true in *s* or both false in *s*

Queste regole sono generalmente semplici, anche se le condizioni di verità per implicazione si discostano in molti casi dalle nostre solite intuizioni del condizionale in inglese. Una formula di forma (P - > Q) è falsa solo quando p è vera e q è falsa. Se p è falsa (dire p corrisponde a la *Luna è fatta di formaggio verde*) e q è vera (dire q corrisponde a *due più due uguale a quattro*) poi P - > q risulterà vera.

Il `LogicParser()` di NLTK analizza espressioni logiche in varie sottoclassi di espressione :

```
>>> lp = nltk.LogicParser()
>>> lp.parse('-(P & Q)')
```

```
<NegatedExpression -(P & Q)>
>>> lp.parse('P & Q')
<AndExpression (P & Q)>
>>> lp.parse('P | (R -> Q)')
<OrExpression (P | (R -> Q))>
>>> lp.parse('P <-> -- P')
<IffExpression (P <-> --P)>
```

Dal punto di vista computazionale, le logiche ci danno un importante strumento per eseguire l'inferenza. Supponiamo di affermare che Freedonia non è al nord di Sylvania e dando come motivazione che Sylvania è a nord della Freedonia. In questo caso, è stato prodotto un **argomento**. La frase *Sylvania è a nord della Freedonia* è la **premessa** dell'argomento mentre *Freedonia è non a nord della Sylvania* è la **conclusione**. Il procedimento che implica il passaggio da una o più premesse ad una conclusione è detto **inferenza**. Informalmente, è comune scrivere gli argomenti in un formato che prevede che la conclusione venga preceduta da *pertanto*.

(9) Sylvania is to the north of Freedonia. Therefore, Freedonia is not to the north of Sylvania. (Sylvania è a nord di Freedonia. Dunque, la Freedonia non è a nord di Sylvania).

Un argomento è **valido** se non c'è nessuna possibile situazione per la quale tutte le premesse sono vere e la conclusione è falsa.

Ora, la validità della (9) dipende fondamentalmente dal significato della frase *a nord di*, in particolare, dal fatto che si tratta di una relazione asimmetrica:

(10) if *x* is to the north of *y* then *y* is not to the north of *x*. (Se *x* è a nord di *y*, allora *y* non è a nord di *x*).

Purtroppo, noi si possono esprimere tali regole in logica proposizionale: i più piccoli elementi con cui abbiamo a che fare sono proposizioni atomiche e noi non possiamo "vederne l'interno" per poter parlare di relazioni tra gli individui *x* e *y*. Il massimo che possiamo fare in questo caso è evidenziare un caso particolare di asimmetria. Utilizziamo il simbolo proposizionale SnF al posto di *Sylvania è a nord della Freedonia* e FnS al posto di *Freedonia è a nord della Sylvania*. Per dire che la *Freedonia non è a nord della Sylvania*, scriviamo FnS - ovvero, trattiamo *non* come equivalente della frase *non è il caso che...*e traduciamolo come l'operatore booleano di un luogo-. Ora possiamo scrivere l'implicazione (10) come segue:

(11) `SnF -> -FnS`

Che ne dite di dare una versione dell'argomentazione completa? Sostituiamo la prima frase di (9) con due formule di logica proposizionale: SnF e anche l'implicazione (11) che esprime (piuttosto male) la nostra conoscenza del significato di *a nord di*. Scriveremo [A1,..., un] / C per rappresentare l'argomento per il quale la conclusione c segue l'ipotesi [A1,..., un]. Questo porta alla seguente rappresentazione dell'argomentazione (9):

(12) `[SnF, SnF -> -FnS] / -FnS`

Questo è un argomento valido: se SnF e SnF - > - FnS sono entrambi veri in una situazione *s*, allora anche - FnS deve essere vero in *s*. Al contrario, se FnS fosse vero, entrerebbero in conflitto con la nostra consapevolezza che due elementi non possono entrambi essere a nord l'uno dell'altro in nessuna situazione possibile. In altre parole, l'elenco `[SnF, SnF - > - FnS, FnS]` è incoerente — queste frasi non possono essere tutte vere insieme.

Gli argomenti possono essere testati per "validità sintattica" utilizzando un sistema di dimostrazione. Approfondiremo questo tema più avanti, nella sezione 10.3.

Le prove logiche possono essere effettuate con il modulo di inferenza di NLTK, ad esempio *tramite* un'interfaccia for example *via* an interface to the third-party theorem prover Prover9.. Gli input al meccanismo di inferenza devono essere analizzati in primo luogo come espressioni logiche dal `LogicParser()`.

```
>>> lp = nltk.LogicParser()
>>> SnF = lp.parse('SnF')
>>> NotFnS = lp.parse('-FnS')
>>> R = lp.parse('SnF -> -FnS')
>>> prover = nltk.Prover9()
>>> prover.prove(NotFnS, [SnF, R])
True
```

Ecco un altro modo di vedere il motivo per cui la conclusione è alla fine. SnF - > - FnS è semanticamente equivalente a - SnF | - FnS, dove "|" è l'operatore biposto corrispondente *all'or*. In generale, ϕ | ψ è vero in una situazione *s* se ϕ è vero in *s* o ϕ è vero in *s*. Ora, supponiamo che sia SnF e - SnF | - FnS siano vere nella situazione *s*. Se SnF è vera, SnF- non può essere vera; un presupposto fondamentale della logica classica è che una frase non può essere vera e falsa allo stesso tempo. Di conseguenza, FnS - deve essere vera.

Ricordiamo che dobbiamo interpretare frasi di un linguaggio logico rispetto a un modello, che è una versione molto semplificata del mondo. Un modello per la logica proposizionale necessità di assegnare i valori True o False a qualsiasi formula possibile. Facciamolo in modo induttivo: primo, ad ogni simbolo proposizionale viene assegnato un valore, poi calcoliamo il valore complessivo delle formule consultando i significati degli operatori boleani (tabella 10.2) e applichiamoli ai valori dei componenti della formula. Una valutazione è un mappatura dalle espressioni di base della logica ai loro valori. Ecco un esempio:

```
>>> val = nltk.Valuation([('P', True), ('Q', True), ('R', False)])
```

Abbiamo inizializzato una valutazione con un elenco di coppie, ognuna delle quali è costituito da un simbolo semantico e da un valore semantico. L'oggetto risultante è essenzialmente solo un dizionario che esegue il mapping di espressioni logiche (trattate come stringhe) in valori appropriati.

```
>>> val['P']
True
```

Come vedremo più avanti, i nostri modelli necessitano di essere un po' più complicati al fine di gestire le forme logiche più complesse discusse nella sezione successiva; per il momento, basta ignorare i parametri dom e g nelle seguenti dichiarazioni.

```
>>> dom = set([])
>>> g = nltk.Assignment(dom)
```

Ora inizializziamo un modello m che utilizza *val* :

```
>>> m = nltk.Model(dom, val)
```

Ogni modello viene fornito con un metodo di valutazione , che determinerà il valore semantico dell'espressione logica, come le formule di logica proposizionale; naturalmente, questi valori dipendono dai valori di verità iniziale che abbiamo assegnato alla proposizionale simboli come `P`, `Q` e `r` .

```
>>> print m.evaluate('(P & Q)', g)
True
>>> print m.evaluate('-(P & Q)', g)
False
>>> print m.evaluate('(P & R)', g)
False
>>> print m.evaluate('(P | R)', g)
True
```

Nota

Tuo turno: Sperimenta con la valutazione diverse formule di logica proposizionale. Il modello fornisce i valori che ti aspettavi?

Fino ad ora, abbiamo tradotto le nostre frasi inglesi in logica proposizionale. Poiché abbiamo il limite di dover rappresentare le frasi atomiche con lettere come p e Q, non possiamo scavare nella loro struttura interna. In effetti, stiamo dicendo che non c'è alcun interesse logico nel suddividere le frasi atomiche in

soggetti, oggetti e predicati dicendo che non c'è nulla di logica interesse a dividere le frasi atomici in soggetti, oggetti e predicati. Tuttavia, questo sembra sbagliato: se si vogliono formalizzare argomenti quali (9), si deve essere in grado di "guardare dentro" le frasi di base. Di conseguenza, andremo al di là della logica proposizionale per arrivare a qualcosa di più espressivo, vale a dire la logica di primo ordine. Questo è ciò di cui si tratterà nella sezione successiva.

10.3 La Logica di primo ordine

Nel seguito di questo capitolo, rappresenteremo il significato delle espressioni del linguaggio naturale traducendole nella logica di primo ordine. Non tutta la semantica del linguaggio naturale può è essere espressa nella logica di primo ordine. In ogni caso è una buona scelta per la semantica computazionale perché è sufficientemente espressivo per sembrare una buona scelta, e d'altra parte, ci sono eccellenti sistemi disponibili per lo svolgimento dell' inferenza automatica nella logica di primo ordine.

Il nostro prossimo passo sarà quello di descrivere come sono costruite le formule di logica di primo ordine, e quindi come tali formule possono essere valutate in un modello.

Sintassi

La logica di primo ordine mantiene tutti gli operatori booleani della logica proposizionale, ma aggiunge alcuni importanti nuovi meccanismi. Per cominciare, le proposizioni sono analizzati in predicati e argomentazioni, il che ci porta un passo più vicini alla struttura delle lingue naturali.

Le regole standard di costruzione per la logica di primo ordine riconoscono i **termini** come singole variabili e costanti individuali, e i **predicati** che assumono differenti numeri di argomenti. Ad esempio, *Angus walks* potrebbe essere formalizzata come *walk(angus)* e *Angus sees Bertie* come *see (angus, bertie)*. Chiameremo Chiediamo *walk* un **predicato unario** e *see* un **predicato binario**. I simboli utilizzati come predicati non hanno un significato intrinseco, anche se è difficile da ricordare. Tornando a uno dei nostri esempi precedenti, non c'è differenza *logica* tra (13a) e (13b).

a. love(margrietje, brunoke)

b. houden_van(margrietje, brunoke)

Di per sé, la logica del primo ordine non ha nulla di effettivo da dire sulla semantica lessicale — il significato di singole parole — ma nonostante questo alcune teorie della semantica lessicale possono essere codificate nella logica di primo ordine. Se un'affermazione atomica come *see (angus, bertie)* è vera o falsa in una situazione non è una questione di logica, ma dipende dalla particolare valutazione che abbiamo scelto per le costanti *see*, *angus* e *bertie*. Per questo motivo, tali espressioni sono chiamate **costanti non-logiche**. Al contrario, le **costanti logiche** (come gli operatori booleani) ricevono sempre la stessa interpretazione in ogni modello per la logica di primo ordine.

Bisogna menzionare il fatto che un predicato binario ha uno status speciale, vale a dire l'uguaglianza, come nelle formule quali *angus = aj*. L'uguaglianza è considerata come una costante logica, poiché per singoli termini *t1* e *t2*, la formula *t1 = t2* è vera se e solo se *t1* e *t2* si riferiscono a una stessa entità.

Spesso è utile analizzare la struttura sintattica delle espressioni della logica di primo ordine, e il modo usuale di fare questo consiste nell'assegnare **tipi** alle espressioni. Seguendo la tradizione della grammatica Montague, useremo due **tipi fondamentali**: *e* è il tipo di entità, mentre *t* rappresenta il tipo di formule, vale a dire, le espressioni che hanno valori di verità. Tenendo conto di questi due tipi fondamentali, si possono formare **tipi complessi** per le espressioni di funzione. Ovvero, dato un qualsiasi tipo σ e τ, <σ, τ> è un tipo complesso corrispondente alle funzioni da 'cose di σ' a 'cose di τ'. Per esempio, <*e*, *t*> è il tipo di espressioni che vanno dall'entità ai valori di verità, chiamati predicati unari. Il LogicParser può essere richiamato in modo che possa occuparsi del controllo del tipo.

```
>>> tlp =
nltk.LogicParser(type_check=True)

>>> parsed =
tlp.parse('walk(angus)')

>>> parsed.argument

<ConstantExpression angus>

>>> parsed.argument.type

e

>>> parsed.function

<ConstantExpression walk>

>>> parsed.function.type

<e,?>
```

Perché vediamo < e,? > alla fine di questo esempio? Anche se il controllore del tipo tenterà di dedurre più tipi possibili, in questo caso esso non è riuscito a specificare in modo completo il tipo di walk, dal momento che il risultato-tipo è sconosciuto. Nonostante si voglia intendere camminare per ricevere il tipo < e, t >, fino a quanto il controllore-tipo conosce, in questo contesto potrebbe essere di altro tipo, come < e, e > o < e, < e, t >. Per aiutare il controllore-tipo, abbiamo bisogno di specificare una **firma**, implementata come un dizionario che associa in modo esplicito i tipi con costanti-non logiche:

```
>>> sig = {'walk': '<e, t>'}

>>> parsed = tlp.parse('walk(angus)', sig)
```

```
>>> parsed.function.type
<e,t>
```

Un predicato binario ha il tipo ⟨*e*, ⟨*e*, *t*⟩⟩. Sebbene questo tipo di qualcosa prima si combina con un argomento di tipo *e* per realizzare un predicato unario, noi rappresentiamo i predicati binari come la diretta combinazione con i loro due argomenti. Ad esempio, il predicato *see* nella traduzione di *Angus see Cyril* si combinerà con i relativi argomenti per dare il risultato *see (angus, cyril)*.

Nella logica di primo ordine, gli argomenti dei predicati possono essere singole variabili *x*, *y* e *z*. In NLTK, adottiamo la convenzione che le variabili di tipo *e* sono tutte minuscole. Le singole variabili sono simili a pronomi personali come *he*, *she* e *it*, in quanto abbiamo bisogno di conoscere il contesto d'uso al fine di capire la loro denotazione.

Un modo di interpretare il pronome in (14) consiste nel riferirsi ad un individuo rilevante nel contesto locale.

(14) He disappeared.

Un altro modo consiste nel fornire un antecedente testuale per il pronome *he*, ad esempio a pronunciare (15°) prima di (14). Qui, diciamo che *he* è **coreferenziale** con il sintagma *Cyril*. Di conseguenza, (14) è semanticamente equivalente a (15b).

(15)

a. Cyril is Angus's dog.

b. Cyril disappeared.

Al contrario si consideri il verificarsi di *lui* in (16a) In questo caso, esso è **vincolato** dall'indefinito NP *un cane*, e questo è un rapporto diverso rispetto alla coreferenza. Se sostituiamo il pronome *egli* con *un cane*, il risultato(16b) *non è* semanticamente equivalente a (16a).

(16)

a. Angus had a dog but he disappeared.

b. Angus had a dog but a dog disappeared.

Corrispondente a (17a) possiamo costruire un **formula aperta** (17b) con due verifiche della variabile *x*. (Si ignori la propensione a semplificare l'esposizione).

(17)

a. He is a dog and he disappeared.

b. dog(x) ∧ disappear(x)

Inserendo un **quantificatore esistenziale** □*x* ('per alcuni *x*') davanti a (17b) si possono **associare** queste variabili, come in (18a) che significa (18b), o il più idiomatico (18c).

(18)

a. □x.(dog(x) ∧ disappear(x))

b. At least one entity is a dog and disappeared.

c. A dog disappeared.

Il rendering di NLTK di (18a)

(19) exists x.(dog(x) & disappear(x))

Oltre al quantificatore esistenziale, la logica di primo ordine ci offre il **quantificatore universale** □*x* ('per ogni *x*'), illustrato nell'esempio (20).

(20)

a. □x.(dog(x) → disappear(x))

b. Everything has the property that if it is a dog, it disappears.

c. Every dog disappeared.

La sintassi NLTK per (20a)

(21) all x.(dog(x) -> disappear(x))

Nonostante (20a) sia la traduzione standard di logica di primo ordine, le condizioni di verità non sono necessariamente quelle che si aspettavano. La formula dice che *se* qualche *x* è un cane, poi *x* scompare— ma non dichiara la presenza di eventuali cani. Quindi, in una situazione in cui non ci sono cani, (20 a)

risulterà ancora vero. (Ricordate che (P - > Q) è vera se p è falsa.) Ora si potrebbe obiettare che *ogni cane scomparso* presupponga l'esistenza di cani, e che la formalizzazione logica è semplicemente errata. Ma è possibile trovare anche altri casi che mancano di tale presupposto. Per esempio, si potrebbe spiegare che il valore dell'espressione Python `astring.replace ('ate', '8')` sia il risultato della sostituzione di tutte le occorrenze di `"ate"` in `astring` da `'8'`, anche se in realtà potrebbe non esserci nessuna di tali occorrenze (tabella 3.2)

Abbiamo visto una serie di esempi dove le variabili sono vincolate da quantificatori. Cosa succede nelle formule come quelle seguenti?

```
((exists x. dog(x)) -> bark(x))
```

The scope of the `exists x` quantifier is `dog(x)`, so the occurrence of `x` in `bark(x)` is unbound. Di conseguenza può essere vincolato a qualsiasi altro quantificatore, ad esempio ogni x nella formula seguente:

```
all x.((exists x. dog(x)) -> bark(x))
```

In generale, un'occorrenza di una variabile x in una formula ϕ è **libera** in ϕ se quell'occorrenza non rientra nell'ambito della x tutti o alcuni x in ϕ. al contrario, se x è libera nella formula ϕ, allora essa è **legata** in x tutti. Φ ed esiste x. Φ. S tutte le variabili delle occorrenze in una formula sono legate, la formula è definita **chiusa**.

Abbiamo accennato prima che il metodo Parse del LogicParser di NLTK restituisce gli oggetti della classe `Expression`. Ogni istanza `expr` di questa classe viene fornita con un metodo libero che restituisce il set di variabili che sono libere in `expr` .

```
>>> lp = nltk.LogicParser()
>>> lp.parse('dog(cyril)').free()
set([])
>>> lp.parse('dog(x)').free()
set([Variable('x')])
>>> lp.parse('own(angus, cyril)').free()
set([])
>>> lp.parse('exists x.dog(x)').free()
set([])
>>> lp.parse('((some x. walk(x)) -> sing(x))').free()
```

```
set([Variable('x')])
>>> lp.parse('exists x.own(y, x)').free()
set([Variable('y')])
```

Dimostrazione del Teorema di Primo Ordine

Richiamiamo il vincolo *to the north of (a nord di)* che abbiamo proposto all'inizio come (10):

(22) if x is to the north of y then y is not to the north of x.

Abbiamo osservato che la logica proposizionale non è sufficientemente espressiva per rappresentare le generalizzazioni dei predicati binari, e di conseguenza non abbiamo ben appreso l'argomento *Sylvania è a nord della Freedonia. Pertanto, la Freedonia non è a nord della Sylvania*.

Si sarà senza dubbio realizzato che la logica di primo ordine, invece, è ideale per la formalizzazione di tali regole:

all x. all y.(north_of(x, y) -> -north_of(y, x))

Ancora meglio, possiamo eseguire un'inferenza automatica per dare prova della validità dell'argomento.

Il caso generale, nell'ambito della dimostrazione del teorema, è determinare se una formula che vogliamo verificare (l'**obiettivo della prova**) può derivare da una sequenza finita di passaggi di inferenza provenienti da un elenco di formule presunte. Scriviamo questo come $s \vdash g$, dove s è un elenco (possibilmente vuoto) di ipotesi e g è un obiettivo di prova. Illustreremo questo esempio utilizzando l'interfaccia di NLTK per la theorem prover, Prover9. In primo luogo, analizziamo l'obiettivo di prova richiesto e le due ipotesi. Poi creiamo un'istanza Prover9 e chiamiamola `prove()` metodo sull'obiettivo, dato l'elenco delle ipotesi.

```
>>> NotFnS = lp.parse('-north_of(f, s)')
>>> SnF = lp.parse('north_of(s, f)')
>>> R = lp.parse('all x. all y. (north_of(x, y) -> -north_of(y, x))')
>>> prover = nltk.Prover9()
>>> prover.prove(NotFnS, [SnF, R])
True
```

Fortunatamente, il theorem prover concorda con noi che l'argomento è valido. Al contrario, si conclude che non è possibile dedurre `north_of(f, s)` dalle nostre ipotesi:

```
>>> FnS = lp.parse('north_of(f, s)')
>>> prover.prove(FnS, [SnF, R])
False
```

Riassumendo il linguaggio della logica di primo ordine

Cogliamo l'opportunità di ribadire le nostre prime regole sintattiche per la logica proposizionale ed aggiungere le regole di formazione per i quantificatori; insieme, questi due elementi ci danno la sintassi della logica di primo ordine. Inoltre, esplicitiamo i tipi di espressioni coinvolte. Adotteremo la convenzione secondo cui $\langle e^n, t \rangle$ è il tipo di un predicato che combina con *n* argomenti di tipo *e* di cedere un'espressione di tipo *t*. In questo caso, diciamo che *n* è il **grado** del predicato.

- Se *p* è un predicato di tipo $\langle e^n e\ t \rangle$ $\alpha_1,\ldots A_n$ sono termini di tipo *e*, quindi $P(\alpha_1,\ldots A_n)$ è di tipo *t*.
- Se α e β sono sia di tipo *e*, quindi (α = β) e (α! = β) sono di tipo *t*.
- Se ϕ è di tipo *t*, così è ϕ -.
- Se ϕ e ψ sono di tipo *t*, quindi sono così (ϕ & ψ), (ϕ | ψ), (ϕ -> ψ) e (ϕ <> - ψ).
- Se ϕ è di tipo *te x* è una variabile di tipo *e*, allora esiste x. Φ e x tutti. Φ sono di tipo *t*.

La Tabella 10.3 riassume le nuove logiche costanti del modulo logica e due dei metodi di Expressions.

Tabella 10.3:

Sintesi delle nuove relazioni logiche e gli operatori richiesti per la logica del primo ordine, insieme con due metodi utili della classe espressione .

Esempio	Descrizione
=	uguaglianza
!=	disuguaglianza
esiste	Quantificatore esistenziale
tutti i	quantificatore universale
e.Free()	Visualizza le variabili libere di e
e.Simplify()	effettuare β-riduzione su e

La Verità nel modello

Abbiamo visto la sintassi della logica di primo ordine, e nella sezione 10.4 esamineremo il compito di tradurre l'inglese nella logica di primo ordine. Come abbiamo già sostenuto nella prima sezione, questo ci viene ulteriormente incontro solo se possiamo dare un significato alle frasi di logica di primo ordine. In altre parole, abbiamo bisogno di dare una *verità condizionale semantica* alla logica di primo ordine. Dal punto di vista della semantica computazionale, ci sono evidenti limiti sulla capacità di questo approccio di spingersi lontano. Sebbene si voglia parlare della veridicità o falsità delle frasi in determinate situazioni, le situazioni stesse possono essere rappresentate nel computer in modo simbolico. Nonostante questa limitazione, è ancora possibile ottenere un quadro più chiaro della verità condizionale semantica codificando i modelli in NLTK.

Dato un linguaggio di logica di primo ordine *L*, un modello *m* per *l* è una coppia 〈*D*, *Val*〉 , dove *D* è un insieme non vuoto chiamato **dominio** del modello e *Val* è una funzione chiamata **funzione di valutazione** che assegna valori da *d* a espressioni di *l* come segue:

- 1. Per ogni singolo costante *c* l, *Val(c)* è un elemento di *D*.
- 2. Per ogni simbolo predicato *P* di grado $n \geq 0$, *Val(P)* è una funzione da d^n {*True*, *False*}. (Se il grado di *p* è 0, allora *Val(P)* è semplicemente un valore di verità, la *P* è considerata come un simbolo proposizionale.)

Secondo (ii), se *p* è di grado 2, quindi *Val(P)* sarà una funzione *f* derivata dalle coppie di elementi di *D* {*True*, *False*}. Nei modelli che dobbiamo costruire in NLTK, adotteremo un'alternativa più conveniente, in cui *Val(P)* è un insieme *s* di coppie, definita come segue:

(23)		$S = \{s \mid f(s) = True\}$

L'elemento *f* è chiamato **funzione caratteristica** di *S* (come discusso in ulteriori letture).

Le relazioni sono semanticamente rappresentate in NLTK nel modo standard di insiemistica: come set di tuple. Ad esempio, supponiamo di avere un dominio di discorso costituito dagli individui Bertie, Olive e Cirillo, dove Bertie è un ragazzo, Olive è una ragazza e Cyril è un cane. Per motivi mnemonici, usiamo b, o e c come le etichette corrispondenti nel modello. Possiamo dichiarare il dominio come segue:

```
>>> dom = set(['b', 'o', 'c'])
```

Usiamo la funzione di utilità `parse_valuation()` per convertire un elenco di stringhe del modulo *`simbolo => valore`* in un oggetto di valutazione .

```
>>> v = """
```

```
... bertie => b
... olive => o
... cyril => c
... boy => {b}
... girl => {o}
... dog => {c}
... walk => {o, c}
... see => {(b, o), (c, b), (o, c)}
... """
>>> val = nltk.parse_valuation(v)
>>> print val
{'bertie': 'b',
 'boy': set([('b',)]),
 'cyril': 'c',
 'dog': set([('c',)]),
 'girl': set([('o',)]),
 'olive': 'o',
 'see': set([('o', 'c'), ('c', 'b'), ('b',
'o')]),
 'walk': set([('c',), ('o',)])}
```

Quindi, secondo la presente stima, il valore di see è un set di tuple tale che Bertie vede Olive, Cyril vede Bertie, e Olive vede Cyril.

Nota

Il tuo turno: disegna un'immagine del dominio di m e i set corrispondenti a ciascuno dei predicati unari, in analogia con il diagramma mostrato nella Figura 10.2.

Avrai notato che i nostri predicati unari (vale a dire, ragazzo, ragazza, cane) also come out as sets of singleton tuples, rather than just sets of individuals. Questa è una convenienza che ci permetterà di avere un trattamento uniforme delle relazioni di qualsiasi grado. Una predicazione del modulo $P(\tau_1,\ldots T_n)$, dove *p*

è di grado *n*, risulta vera nel caso in cui la tupla di valori corrispondenti a (τ_1,... T_n) appartiene al set di tuple nel valore di *p*.

```
>>> ('o', 'c') in val['see']
True
>>> ('b',) in val['boy']
True
```

Le singole variabili e le assegnazioni

Nei nostri modelli, la controparte di un contesto d'uso è un'**assegnazione di variabile**. Questa è un mappatura che, nel dominio, va dalle singole variabili alle entità. Le assegnazioni vengono create utilizzando il costruttore di assegnazione , che considera come paramentro anche il dominio del modello. Non ci è richiesto di entrare in merito alle associazioni, ma se lo facessimo, sarebbero in un formato (*variabile*, *valore*) simile a quello che abbiamo visto in precedenza per le valutazioni.

```
>>> g = nltk.Assignment(dom, [('x', 'o'),
('y', 'c')])
>>> g
{'y': 'c', 'x': 'o'}
```

Inoltre, c'è un formato `print()` per le assegnazioni che utilizza una notazione più vicina a quella che spesso si trova nei libri di logica:

```
>>> print g
g[c/y][o/x]
```

Vediamo come valutare una formula atomica di logica di primo ordine. In primo luogo, si crea un modello, poi si utilizza il metodo `evaluate ()` per calcolare il valore di verità.

```
>>> m = nltk.Model(dom, val)
>>> m.evaluate('see(olive, y)', g)
```

```
True
```

Che cosa sta accadendo? Si sta valutando una formula simile al nostro esempio precedente, vedere (olive, cyril). Tuttavia, quando la funzione di interpretazione incontra la variabile y, piuttosto che cercare un valore in val, chiede l'assegnazione di variabile g per ottenere un valore:

```
>>> g['y']
'c'
```

Poiché sappiamo che gli individui o e c stanno nel rapporto *see* , il valore che ci aspettavamo è True. In questo caso, possiamo dire che l'assegnazione g **soddisfa** la formula v. (olive, y). Al contrario, la formula seguente restituisce False in relazione a g —

```
>>> m.evaluate('see(y, x)', g)
False
```

Nel nostro approccio (anche se non nella logica standard di primo ordine), le assegnazioni delle variabili sono *parziali*. Ad esempio, g non dice nulla su tutte le variabili x e y. Il metodo `purge()` cancella tutte le associazioni da un'assegnazione.

```
>>> g.purge()
>>> g
{}
```

Se cerchiamo ora di valutare una formula come `see (olive, y)` relativa a g, è come se cercassimo di interpretare una frase contenente un *lui* quando non sappiamo a cosa *he* si riferisca. In questo caso, la funzione di valutazione non riesce a fornire un valore di verità.

```
>>> m.evaluate('see(olive, y)', g)
'Undefined'
```

Dal momento in cui i nostri modelli contengono già le regole per l'interpretazione degli operatori booleani, possono essere composte e valutate, arbitrariamente, delle formule complesse.

```
>>> m.evaluate('see(bertie, olive) & boy(bertie) & -walk(bertie)', g)
True
```

Il processo generale che determina la veridicità o la falsità di una formula in un modello è detto **model checking**.

Quantificazione

Una delle intuizioni cruciali della logica moderna è che la nozione di soddisfazione della variabile può essere utilizzata per fornire un'interpretazione delle formule quantificate. Utilizziamo il punto (24) come esempio.

(24)		`exists x.(girl(x) & walk(x))`

Questo quando risulta vero? Pensiamo a tutti gli individui nel nostro dominio, vale a dire, nel `dom`. Vogliamo verificare se uno qualsiasi di questi individui possiede le proprietà di essere una ragazza e di camminare. In altre parole, vogliamo sapere se c'è qualche *u* nel `dom` tale per cui `g [u / x]` soddisfa la formula aperta (25).

(25)		`Girl(x) & walk(x)`

Si tenga presente quanto segue:

```
>>> m.evaluate('exists x.(girl(x) & walk(x))', g)
True
```

`Evaluate ()` restituisce True perché qui c'è qualche *u* nel dom tale che (25) è soddisfatta da un'assegnazione che lega x a *u*. In realtà, o è tale e quale ad *u*:

```
>>> m.evaluate('girl(x) & walk(x)', g.add('x', 'o'))
True
```

Un utile strumento offerto da NLTK è il metodo `satisfiers()` . Quest'ultimo restituisce un insieme di tutti gli individui che soddisfano una formula aperta. I parametri del metodo sono una formula analizzata, una variabile e un'assegnazione. Ecco alcuni esempi:

```
>>> fmla1 = lp.parse('girl(x) | boy(x)')
>>> m.satisfiers(fmla1, 'x', g)
set(['b', 'o'])
>>> fmla2 = lp.parse('girl(x) -> walk(x)')
>>> m.satisfiers(fmla2, 'x', g)
set(['c', 'b', 'o'])
>>> fmla3 = lp.parse('walk(x) -> girl(x)')
>>> m.satisfiers(fmla3, 'x', g)
set(['b', 'o'])
```

È utile riflettere sul perchè fmla2 e fmla3 riportano questi valori. Le condizioni di verità per -> significa che fmla2 equivale a -girl(x) | walk(x), che è soddisfatta da qualcosa che non è una ragazza o una camminata. Dal momento che né b (Bertie) né c (Cyril) sono ragazze, secondo il modello m, entrambi soddisfano l'intera formula. Naturalmente o soddisfa la formula perché o soddisfa entrambe le disgiunzioni. Ora, dal momento che ogni membro del dominio del discorso soddisfa fmla2, anche la corrispondente formula universalmente quantificata è vera.

```
>>> m.evaluate('all x.(girl(x) -> walk(x))', g)
True
```

In altre parole, una formula universalmente quantificata *x*..ϕ è vera rispetto a g nel caso in cui per ogni *u*, ϕ è vera rispetto a g [u / x] .

Nota

Il tuo turno: Prova a trovare, prima utilizzando carta e penna e solo in seguito m.evaluate(), quali sono i valori di verità per all i x.(girl(x) e walk(x)) e exsists x.(boy(x) - > walk(x)). Fate in modo di capire perché ricevono tali valori.

Il quantificatore di portata dell'ambiguità

Che cosa succede quando si da una rappresentazione formale di una frase con *due* quantificatori, come la seguente?

(26)		Everybody admires someone. (Tutti ammirano qualcuno)

Ci sono (almeno) due modi per esprimere (26) secondo la logica di primo ordine:

(27)

a. all x.(person(x) -> exists y.(person(y) & admire(x,y)))

b. exists y.(person(y) & all x.(person(x) -> admire(x,y)))

Si possono usare entrambi? La risposta è sì, ma essi hanno significati diversi. (27b) è logicamente più forte di (27 a) esso sostiene che c'è una persona sola, diciamo Bruce, che è ammirata da tutti. (27 a) invece, richiede solo che per ogni persona *u*, si trovi qualche persona *u'* che *u* ammira; ma questa potrebbe essere una persona diversa *u'* in ogni caso (27 a) e (27b) vengono distinti in termini di **portata** dei quantificatori. Nel primo, ha una portata più ampia rispetto a mentre in (27 b) l'ordinazione di portata è invertita. Così ora abbiamo due modi di rappresentare il significato di (26) e sono entrambi legittimi. In altre parole, stiamo sostenendo che (26) è *ambiguo* rispetto alla portata del quantificatore, e le formule (27) ci danno un modo per rendere esplicite le due letture. Tuttavia, non siamo solo interessati ad associare due rappresentazioni distinte (26). Vogliamo anche visualizzare in dettaglio come le due rappresentazioni portano a diverse condizioni di verità in un modello.

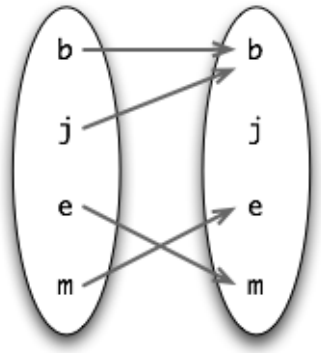
Per poter esaminare l'ambiguità più da vicino, fissiamo la seguente valutazione:

```
>>> v2 = """
... bruce => b
... cyril => c
... elspeth => e
... julia => j
```

```
... matthew => m
... person => {b, e, j, m}
... admire => {(j, b), (b, b), (m, e), (e, m), (c, a)}
... """
>>> val2 = nltk.parse_valuation(v2)
```

La relazione *admire* può essere visualizzata utilizzando il diagramma di mappatura mostrato nella (28):

(28)



La freccia tra due individui *x* e *y* indica che *x* ammira *y*. Perciò j e b ammirano b (Bruce è molto vanitoso), mentre e ammira m e m ammira e. In questo modello, la formula, prima menzionata, (27 a) è vera ma (27b) è falsa. Un modo di esplorare questi risultati consiste nell'utilizzo del metodo satisfiers() degli oggetti del modello .

```
>>> dom2 = val2.domain
>>> m2 = nltk.Model(dom2, val2)
>>> g2 = nltk.Assignment(dom2)
>>> fmla4 = lp.parse('(person(x) -> exists y.(person(y) & admire(x,
y)))')
>>> m2.satisfiers(fmla4, 'x', g2)
set(['a', 'c', 'b', 'e', 'j', 'm'])
```

Questo dimostra che fmla4 detiene il dominio di ogni individuo. Al contrario, si consideri la formula fmla5 qui sotto; questa non ha nessun elemento di soddisfazione per la variabile y .

```
>>> fmla5 = lp.parse('(person(y) & all x.(person(x) ->
admire(x, y)))')
```

```
>>> m2.satisfiers(fmla5, 'y', g2)
set([])
```

Vale a dire, non c'è nessuna persona che è ammirata da tutti. Prendendo una diversa formula aperta, fmla6, possiamo verificare che c'è una persona, chiamata Bruce, che è ammirata da Julia e Bruce.

```
>>> fmla6 = lp.parse('(person(y) & all x.((x = bruce | x =
julia) -> admire(x, y)))')
>>> m2.satisfiers(fmla6, 'y', g2)
set(['b'])
```

Nota

Il tuo turno: Elabora un nuovo modello basato su m2 tale che (27a) esca falsa nel tuo modello; allo stesso modo, elabora un nuovo modello in modo tale che (27b) esca vera.

Costruzione del modello

Abbiamo assunto di avere già un modello all'interno del quale si è indagata la veridicità di una frase. Al contrario, la costruzione del modello, tenta di creare un nuovo modello, dato un insieme di frasi. Se riesce in ciò, vuol dire che l'insieme è coerente, dal momento in cui abbiamo una prova dell'esistenza del modello.

Evochiamo il generatore di modello Mace4, creando un'istanza di Mace() e chiamando il relativo metodo build_model().Un'opzione è trattare il nostro set di frasi come assunzioni, lasciando l'obiettivo non specificato. L'interazione seguente mostra come sia [a, c1] chee [a, c2] sono liste coerente, dal momento che Mace riesce a costruire un modello per ciascuno di essi, mentre [c1, c2] è incoerente.

```
>>> a3 = lp.parse('exists x.(man(x) & walks(x))')
>>> c1 = lp.parse('mortal(socrates)')
>>> c2 = lp.parse('-mortal(socrates)')
>>> mb = nltk.Mace(5)
>>> print mb.build_model(None, [a3, c1])
True
>>> print mb.build_model(None, [a3, c2])
```

```
True
>>> print mb.build_model(None, [c1, c2])
False
```

Il generatore di modelli può essere anche utilizzato come aggiunta al theorem prover. Supponiamo di voler dimostrare che s ⊢ g, vale a dire che g è logicamente ricavabili dalle ipotesi S = [s1, s2,..., sn]. Possiamo dare questo stesso input a Mace4 e il generatore di modelli cercherà di trovare un contro-esempio, utile a mostrare che g *non* proviene da S. Così, data questa direttiva, Mace4 cercherà di trovare un modello per il set s contemporaneamente alla negazione di g, chiamando la lista S' = [s1, s2,..., sn, -g]. Se g non riesce ad essere dimostrato da S, Mace4 potrebbe tornare utile con un controesempio più veloce rispetto a Prover9 che ha concluso di non poter trovare la prova richiesta. If g fails to follow from S, then Mace4 may well return with a counterexample faster than Prover9 concludes that it cannot find the required proof Al contrario, se g *è* dimostrabile da S, Mace4 potrebbe spendere molto tempo cercando invano di trovare un contro-modello e alla fine rinunciare.

Consideriamo uno scenario concreto. La nostra ipotesi sono l'elenco [C'è *una donna che è amata da ogni uomo*, *Adam è un uomo*, *Eva è una donna*]. La nostra conclusione è che *Adam ama Eve*. Mace4 può trovare un modello nel qual le premesse sono vere ma la conclusione è falsa? Nel codice seguente, usiamo MaceCommand() che ci permetterà di controllare il modello che è stato costruito.

```
>>> a4 = lp.parse('exists y. (woman(y) & all x. (man(x) ->
love(x,y)))')
>>> a5 = lp.parse('man(adam)')
>>> a6 = lp.parse('woman(eve)')
>>> g = lp.parse('love(adam,eve)')
>>> mc = nltk.MaceCommand(g, assumptions=[a4, a5, a6])
>>> mc.build_model()
True
```

Così la risposta è sì: Mace4 ha trovato un contro-modello nel quale esiste qualche altra donna, oltre ad Eva, che Adamo ama. Ma diamo un'occhiata più da vicino al modello di Mace4, convertito nel formato che usiamo per le valutazioni.

```
>>> print mc.valuation
```

```
{'C1': 'b',
 'adam': 'a',
 'eve': 'a',
 'love': set([('a', 'b')]),
 'man': set([('a',)]),
 'woman': set([('a',), ('b',)])}
```

La forma generale di questa valutazione dovrebbe essere familiare: essa contiene alcune costanti individuali e predicati, ciascuno con un appropriato tipo di valore. Ciò che potrebbe essere sconcertante è il valore C1. Questo è una "costante skolem" che il generatore di modello introduce come rappresentante del quantificatore esistenziale. Cioè, quando sopra il generatore di modello ha rilevato che exsists parte di y a4, sapeva che c'è qualche individuo b nel dominio che soddisfa la formula aperta nel corpo della a4. Tuttavia, non sa se b è anche la denotazione di una costante individuale che si trova chissà dove nel suo input, così costituisce al momento un nuovo nome per b, vale a dire C1. Ora, dal momento che le nostre premesse non hanno detto nulla circa le costanti individuali Adamo ed Eva, il generatore di modello ha deciso che non c'è alcun motivo per trattarli come entità denotanti diverse, ed entrambi vengono mappati ad a. Inoltre, poiché non abbiamo specificato che l'uomo e la donna denotano insiemi disgiunti, il generatore consente che le denotazioni si sovrappongano. Questo dimostra in modo abbastanza drastico la conoscenza implicita che ci supporta nell'interpretazione del nostro scenario, ma della quale il generatore di modelli non sa nulla. Per questo motivo aggiungiamo un nuovo assunto per rendere l'insieme uomini e l'insieme donne disgiunti. Il generatore di modello produce di nuovo un contro-modello ma questa volta è più in linea con le nostre intuizioni sulla situazione:

```
>>> a7 = lp.parse('all x. (man(x) -> -woman(x))')
>>> g = lp.parse('love(adam,eve)')
>>> mc = nltk.MaceCommand(g, assumptions=[a4, a5, a6, a7])
>>> mc.build_model()
True
>>> print mc.valuation
{'C1': 'c',
 'adam': 'a',
 'eve': 'b',
 'love': set([('a', 'c')]),
 'man': set([('a',)]),
 'woman': set([('b',), ('c',)])}
```

Riflettendoci, si può vedere che non c'è niente nelle nostre premesse che afferma che Eva è l'unica donna nel dominio del discorso, e per questo motivo il contro-modello, infatti, è accettabile. Se avessimo voluto renderlo fuori regola, avremmo dovuto aggiungere un ulteriore presupposto, del tipo `exsists y. x. all (woman(x) - > (x = y))` garantendo, così, la presenza di una sola donna nel modello.

10.4 La semantica delle frasi inglesi

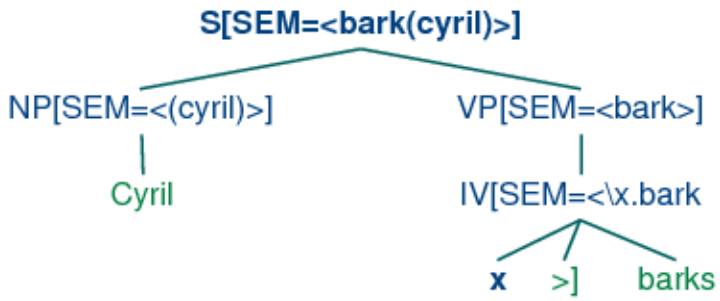
La semantica composizionale nella grammatica basata sulle funzionalità

All'inizio del capitolo abbiamo brevemente illustrato un metodo di costruzione di rappresentazioni semantiche sulla base di un'analisi sintattica, utilizzando il framework di grammatica sviluppato nel capitolo 9. Questa volta, piuttosto che costruire una query SQL, costruiremo una forma logica. Una delle nostre idee guida per la progettazione di tali grammatiche è il Principio di composizionalità . (Noto anche come principio di Frege; vedere (Gleitman & Liberman,1955) per la formulazione riportata di seguito.)

Principio di composizionalità: Il significato di un intero è una funzione dei significati delle parti e del modo in cui queste vengono combinate in sintatticamente.

Si supporrà che le parti pertinenti semanticamente di un'espressione complessa siano date da una teoria di analisi sintattica. All'interno di questo capitolo, daremo per scontato che le espressioni sono analizzate in sullo sfondo di un contesto libero dalla grammatica. In ogni caso, questo non dipende dal Principio di composizionalità.

Il nostro obiettivo è di integrare, nel modo più semplice, la costruzione di una rappresentazione semantica con il processo di analisi. La figura (29) illustra una prima approssimazione al tipo di analisi che vorrebbe costruire.



Nella figura il valore di sem al nodo principale mostra una rappresentazione semantica per l'intera frase, mentre i valori di sem ai nodi inferiori mostrano le rappresentazioni semantiche per i componenti della frase. Poiché i valori di sem devono essere trattati in modo speciale, essi si distinguono dagli altri valori della funzione essendo racchiusi tra parentesi angolari.

Fin'ora nessun problema, ma come si scrivono le regole grammaticali che ci danno questo tipo di risultato? Il nostro approccio sarà simile a quello adottato per la grammatica sql0.fcfg all'inizio di questo capitolo, in quanto assegneremo le rappresentazioni semantiche ai nodi lessicali e poi comporremo le rappresentazioni semantiche per ogni frase partendo da quelle dei suoi nodi figlio. In questo caso, tuttavia, piuttosto che la

concatenazione a stringa, utilizzeremo come modalità di composizione l'applicazione di funzione. Per essere più precisi, supponiamo di avere i costituenti NP e VP con i valori appropriati per i loro nodi di sem . Il valore di sem di una s è gestito da una regola come (30). (Osservare che nel caso in cui il valore di sem è una variabile, si omettono le parentesi angolari).

(30)		`S [SEM = <? vp(?np) >]-> NP [SEM =? subj] VP [SEM =? vp]`

La formula ci dice che è stato dato qualche valore sem ? subj per il soggetto NP e qualche valore sem ? vp per il VP, il valore di sem del padre s è costruito mediante l'applicazione di ? vp come un'espressione di funzione ? np. Da questo, possiamo concludere che ? vp indica una funzione che ha la denotazione di ? np nel suo dominio. (30) è un buon esempio di costruzione semantica usando il principio di composizionalità.

Completare la grammatica è molto semplice; tutti ciò di cui abbiamo bisogno sono le regole illustrate di seguito.

```
VP[SEM=?v] -> IV[SEM=?v]

NP[SEM=<cyril>] -> 'Cyril'

IV[SEM=<\x.bark(x)>] -> 'barks'
```

La regola VP dice che la semantica del genitore è la stessa semantica che è a capo di quella del bambino. Le due regole lessicali forniscono costanti non-logiche che servono rispettivamente come i valori semantici di *Cyril* e *barks*. C'è un pezzo di notazione supplementare alla voce *barks* che spiegheremo a breve.

Prima di scendere nei dettagli delle regole di composizione semantica, abbiamo bisogno di aggiungere un nuovo strumento al nostro kit, vale a dire il calcolo λ. Questo ci fornisce uno strumento prezioso per la combinazione di espressioni della logica di primo ordine, come assemblare la rappresentazione del significato di una frase inglese.

Il calcolo di λ

Al punto 1.3 abbiamo evidenziato che la notazione matematica degli insiemi è un metodo utile per specificare le proprietà *p* delle parole che si vogliono selezionare da un documento. Lo abbiamo dimostrato con (31) che abbiamo letto come "l'insieme di tutti i *w* tale che *w* è un elemento di *V* (il vocabolario) e *w* ha la proprietà *p*".

(31)		{*w* | *p* □ *v* & *P(w)*}

Essa risulta estremamente utile per aggiungere qualcosa alla logica di primo ordine che permetterà di conseguire lo stesso effetto. Lo facciamo con l'**operatore λ** (pronunciato "lambda"). La controparte λ (31) è (32). (Dato che qui non si sta cercando di fare la teoria degli insiemi, bisogna solo trattare *v* come un predicato unario.)

(32)		$\Lambda w...\ (V(w) \wedge P(w))$

Nota

Le espressioni λ furono originariamente progettate da Alonzo Church per rappresentare le funzioni calcolabili e per fornire le basi per la matematica e logica. La teoria in cui le espressioni λ sono studiate è conosciuta come il calcolo di λ. Si noti che il calcolo di λ non è parte della logica del primo ordine — entrambi i processi possono essere utilizzati indipendentemente l'uno dall'altro.

Λ è un operatore di associazione, così come lo sono i quantificatori della logica di primo ordine. Se abbiamo una formula aperta come (33 a) allora possiamo associare la variabile *x* con l'operatore λ, come mostrato nella (33b). La corrispondente rappresentazione NLTK è data in (33c).

a.		$(walk(x) \wedge chew\_gum(x))$
b.		$\lambda x.(walk(x) \wedge chew\_gum(x))$
c.		\x.(walk(x) & chew_gum(x))

Ricordate che \ è un carattere speciale nelle stringhe di Python. Questo carattere speciale si può fuggire,chiudere, (con un altro \), o altrimenti utilizzando "raw strings ":

```
>>> lp = nltk.LogicParser()
>>> e = lp.parse(r'\x.(walk(x) & chew_gum(x))')
>>> e
<LambdaExpression \x.(walk(x) & chew_gum(x))>
>>> e.free()
set([])
>>> print lp.parse(r'\x.(walk(x) & chew_gum(y))')
\x.(walk(x) & chew_gum(y))
```

Si usa un nome speciale per il risultato dell'unione delle variabili in un'espressione: **astrazione λ**. Quando si incontrano per la prima volta gli abstract λ, può essere difficile comprendere intuitivamente il senso del loro significato. Un paio di annotazioni per (33b) sono: "essere una *x* tale che *x* cammina e *x* mastica gomma" o "avere la proprietà di camminare e masticare". Spesso è stato suggerito che gli abstract λ sono buone rappresentazioni per le frasi verbali (o proposizioni senza soggetto), in particolare quando queste si presentano come argomenti a sé. Questo è illustrato nell'esempio (34 a) nella sua traduzione (34 b).

(34) a. To walk and chew gums is hard. (Camminare e masticare la gomma è difficile)

```
b.  hard(\x.(walk(x) & chew_gum(x))
```

Il quadro generale è pertanto questo: data una formula aperta u ϕ con la variabile libera *x*, astraendo su *x* si ottiene l'espressione di appartenenza λ*x.*Φ — la proprietà di essere un *x* tale che ϕ. Ecco una versione più ufficiale di come sono costruiti gli abstracts:

(35)		Se α è di tipo τ, e *x* è una variabile di tipo e, quindi \x. A è di tipo 〈*e*, τ〉 .

(34b)ci ha mostrato un caso nel quale diciamo qualcosa circa una proprietà, vale a dire *difficile.* Ma solitamente quello che si fa con le proprietà è attribuirle a degli individui. E in effetti se ϕ è una formula aperta, l'abstract λ*x.*Φ può essere utilizzato come un predicato unario. Negli esempi (36) e (33b) si basa sul termine *gerald*.

(36)		`\X.(Walk(x) & chew_gum(x)) (gerald)`

Ora (36) dice che Gerald ha la proprietà di camminare e di masticare la gomma, il che ha lo stesso significato di (37).

(37		`(walk(gerald) e chew_gum(gerald))`

Quello che abbiamo fatto è stato rimuovere \x fin dall'inizio di \x.(walk(x) & chew_gum(x)) e sostituire tutte le occorrenze di x in (walk(x) / chew_gum(x)) da gerald. Useremo α [β*/x*] come notazione per l'operazione di sostituzione di tutte le occorrenze libere di *x* in α con l'espressione β. Così:

(walk(x) & chew_gum(x))[gerald/x]

è la stessa espressione di (37). La "riduzione" (36) (37) è un'operazione estremamente utile nella semplificazione delle rappresentazioni semantiche e potremo utilizzarla molto nel resto del capitolo. L'operazione è spesso chiamata **riduzione di β.** Affinché sia semanticamente giustificata, vogliamo mantenere per λ*x.* α(β) gli stessi valori semantici di α [β/*x*]. Questo è vero, e soggetto a una lieve complicazione che vedremo tra poco. Al fine di portare la **riduzione di β** delle espressioni in NLTK, possiamo richiamare il metodo simplify() .

```
>>> e = lp.parse(r'\x.(walk(x) & chew_gum(x))(gerald)')
>>> print e
\x.(walk(x) & chew_gum(x))(gerald)
>>> print e.simplify()
(walk(gerald) & chew_gum(gerald))
```

Anche se finora abbiamo considerato solo i casi in cui il corpo dell'abstract λ è una formula aperta, vale a dire, di tipo *t*, questa non è una restrizione necessaria; il corpo può essere qualsiasi espressione ben formata. Ecco un esempio con due λs.

(38)
```
\x.\y.(dog(x) & own(y, x))
```

Proprio come (33b) interpreta il ruolo di un predicato unario, (38) funziona come un predicato binario: può essere applicato direttamente a due argomenti _. Il LogicParser permette ai λs nidificati, ad esempio \x.\y. di essere scritti nella forma abbreviata \x y.

```
>>> print lp.parse(r'\x.\y.(dog(x) & own(y, x))(cyril)').simplify()
\y.(dog(cyril) & own(y,cyril))
>>> print lp.parse(r'\x y.(dog(x) & own(y, x))(cyril, angus)').simplify()
(dog(cyril) & own(angus,cyril))
```

Tutti i nostri abstract λ finora hanno coinvolto le variabili familiari di primo ordine: x, y e così via — le variabili di tipo *e*. Ma supponiamo di voler trattare un abstract, diciamo \x.walk(x) come *argomento* di un altro abstract λ? Si potrebbe tentare questo:

```
\y.y(angus)(\x.walk(x))
```

Poiché,però, è previsto che la variabile y sia di tipo *e*, \y.y(angus) questa vale solo per gli argomenti di tipo *e* mentre \x.walk(x) è di tipo 〈*e*, *t*〉 ! Abbiamo bisogno di permettere l'astrazione su variabili di tipo superiore. Utilizziamo,allora, p e q come variabili di tipo 〈*e*, 〉 *t*, ottenendo un abstract come ad esempio \P.P(angus). Poiché p è di tipo 〈*e*, *t*〉 , l'intero abstract è di tipo 〈 〈*e*, 〉 *t*, *t*〉 . \P.P(angus)(\x.walk(x)) è legale e può essere semplificata tramite β-riduzione a \x.walk(x)(angus) e poi nuovamente a walk(angus)

Durante lo svolgimento della **riduzione di β**, bisogna trattare le variabili con una certa accortezza. Si consideri, ad esempio, i termini λ (39 a) e (39b) che si differenziano solo per l'identità di una variabile libera.

(39)

a.		\y.see(y, x)
b.		\y.see(y, z)

Supponiamo di applicare il termine λ \P.exists x.P(x) a ciascuno di questi termini:

(40)

a.		\P.exists x.P(x)(\y.see(y, x))
b.		\P.exists x.P(x)(\y.see(y, z))

All'inizio abbiamo sottolineato che i risultati dell'applicazione dovrebbero essere semanticamente equivalenti. Ma se lasciamo rientrare la variabile libera x in (39 a) nell'ambito del quantificatore esistenziale in (40 a) a seguito della riduzione, il risultato sarà diverso:

(41)

a.		exists x.see(x, x)
b.		exists x.see(x, z)

(41 a) significa che c'è qualche x che vede se stesso, mentre (41b) significa che c'è qualche x che vede un imprecisato individuo z. Che cosa è andato storto qui? Chiaramente, vogliamo proibire il tipo di variabile "cattura" mostrata in (41 a)

Per ovviare a questo problema, facciamo un passo indietro. Ha importanza quale nome particolare usiamo per la variabile vincolata dal quantificatore esistenziale nell'espressione di funzione (40 a) ? La risposta è No. In realtà, data qualsiasi espressione di associazione variabile (che coinvolge□, □ o λ), il nome scelto per la variabile associata è completamente arbitrario. Ad esempio, x.P(x) ed exsists y.P(y) sono equivalenti; essi sono chiamati **equivalenti di α**, o **varianti alfabetici**. Il processo di rietichettatura delle variabili associate è conosciuto come **conversione di α**. Quando si verifica l'uguaglianza di VariableBinderExpressions nel modulo di logica (vale a dire utilizzando ==), stiamo ricercando l'equivalenza di α:

```
>>> e1 = lp.parse('exists x.P(x)')

>>> print e1

exists x.P(x)

>>> e2 = e1.alpha_convert(nltk.sem.Variable('z'))

>>> print e2

exists z.P(z)

>>> e1 == e2

True
```

Quando la riduzione di β è effettuata su un'applicazione f, controlliamo se esistono variabili libere in a che si verificano anche come variabili associate a qualsiasi termini subordinati di f. Supponiamo, come nell'esempio illustrato in precedenza, che x è libero in a, e che f contiene il termine subordinato exsists x.P(x). In questo caso, si produce una variante alfabetica di exsists x.P(x), dice, exsists z1.P(Z1)e poi vanti con la riduzione. Questa rietichettatura viene effettuata automaticamente dal codice di riduzione di β in logica, e i risultati possono essere visti nell'esempio seguente.

```
>>> e3 = lp.parse('\P.exists x.P(x)(\y.see(y, x))')

>>> print e3

(\P.exists x.P(x))(\y.see(y,x))

>>> print e3.simplify()

exists z1.see(z1,x)
```

Nota

Come negli esempi attraverso cui lavorerai nelle prossime sezioni, troverai che le espressioni logiche che vengono restituite hanno diversi nomi di variabili; per esempio si potrebbe vedere z14 al posto di z1 nella formula di cui sopra. Questo cambiamento di etichettatura è innocuo — in realtà, esso è solo un'illustrazione delle varianti alfabetiche.

Dopo questo excursus, torniamo al compito di costruire forme logiche per le frasi in inglese.

Gli NP quantificati

All'inizio di questa sezione, abbiamo brevemente descritto come costruire una rappresentazione semantica per *Cyril barks*. Sarai perdonato per aver pensato che fosse tutto troppo semplice — sicuramente c'è qualcosa in più per costruire la semantica composizionale. Per esempio, cosa dire riguardo ai quantificatori? Giusto, questo è un problema cruciale. Ad esempio, si vuole che (42 a) venga restituito nella forma logica (42 b). Come avviene ciò?

a.		A dog barks.
b.		exists x.(dog(x) & bark(x))

Ipotizziamo che la nostra operazione *only* per la costruzione di rappresentazioni semantiche complesse sia l'applicazione della funzione. I nostro problema sarà questo: come diamo una rappresentazione semantica al quantificatore NPs *a dog* in modo tale che possa combinarsi con bark e dare il risultato in (42b)? Come primo passo, facciamo agire il valore del soggetto sem come espressione di funzione, piuttosto che come argomento. (Questo procedimento a volte è chiamato **type-raising,** raccolta dei tipi.) Ora stiamo cercando un modo per rendere un'istanza ? np in modo che [SEM = <? np(\x.bark(x)) >] equivalga a [SEM = < esiste x.(dog(x) & bark(x)) >]. Non sembra ricordare lontanamente l'effettuazione della riduzione di β nel calcolo di λ? In altre parole, ci vuole un termine λ *m* per sostituire ? np cosicché applicando *m* a 'bark' risulti (42b). A tale scopo, sostituiamo 'bark' in (42b) con una variabile predicativa 'P'e leghiamo la variabile con λ, come mostrato nella (43)

```
(43) \P.exists x.(dog(x) & P(x))
```

Abbiamo usato un diverso stile di variabile nell'esempio(43) — cioè 'P' piuttosto che ' *x* ' o 'y' — per segnalare che noi stiamo astraendo su un diverso tipo di oggetto — non un individuo, ma un'espressione di funzione di tipo 〈*e*, 〉 *t*. Così il (43) è come un intero 〈 〈*e*, 〉 *t*, *t*〉 . Prenderemo questo come tipo di NPs in generale. Evidenziandolo ulteriormente, un quantificatore universale NP avrà un aspetto simile (44).

(44)		\P.all x.(dog(x) - > P(x))

Anche se abbiamo fatto abbastanza ora, vogliamo ancora effettuare un ulteriore astrazione più applicazione per il processo di combinazione semantica della determinante *a* vale a dire (46) con la semantica *dog*.

```
(45) \Q P.exists x.(Q(x) & P(x))
```

Applicando (46) come l'espressione della funzione dog viene prodotta (43) e applicandolo a bark ci dà `\P.exists x.(dog(x) & P(x))(\x.bark(x))`. Infine, svolgendo la riduzione di β produce solo quello che volevamo, vale a dire (42 b).

Verbi transitivi

La nostra prossima sfida è quella di affrontare le frasi contenenti i verbi transitivi, come ad esempio (46)

(46)		Angus chases a dog. (Angus insegue un cane)

La semantica di uscita che vogliamo costruire è `exsists x.(dog(x) & chase (angus, x))`. Diamo un'occhiata a come si possa utilizzare l'astrazione di λ per ottenere questo risultato. Un vincolo significativo sulle possibili soluzioni consiste nel richiedere che la rappresentazione semantica di *a dog* sia indipendente se la NP agisce come soggetto o l'oggetto della frase. In other words, we want to get the formula above as our output while sticking to (43) the NP semantics. Un secondo vincolo è che i VP devono avere un tipo uniforme di interpretazione, indipendentemente se siano costituiti da un solo verbo intransitivo o da un verbo transitivo più un oggetto. In particolare, si prevede che i VP siano sempre di tipo 〈*e*, 〉 *t*. Dati questi vincoli, ecco una rappresentazione semantica per *chases a dog* che mette in atto il trucchetto.

(47)		`\Y.Exists x.(dog(x) & chase (y, x))`

Pensiamo a (47) come la proprietà di essere un *y* tale che per qualche cane *x*, *y* insegue *x*; o più colloquialmente, che sia una *y* che insegue un cane. Il nostro compito consiste nel progettare una

rappresentazione semantica per *chases* che si possa combinare con (43), così da permettere a (47) di esserne un derivato.

Svolgiamo l'inverso della riduzione di β su (47) dando origine a (48).

(48)		`\P.Exists x.(dog(x) / P(x)) (\z.chase (y, z))`

(48) può essere un po' difficile da leggere in un primo momento; you need to see that it involves applying the quantified Np representation from (43) to \z.chase(y,z). (48) is equivalent via β-reduction to exists x.(dog(x) & chase(y, x)).

Ora cerchiamo di sostituire l'espressione di funzione (48) con una variabile x dello stesso tipo, come un NP; che è, di tipo 〈〈*e*,〉 *t*, *t*〉 .

(49)		`X (\z.chase (y, z))`

La rappresentazione di un verbo transitivo dovrà applicare a un argomento del tipo x per produrre un'espressione di funzione del tipo di VPs, cioè, di tipo 〈*e*,〉 *t*. Possiamo assicurare questo applicando l'astrazione sia sulla variabile X (49) che sulla variabile oggetto y. In questo modo la soluzione completa viene raggiunta dando a *chases* la rappresentazione semantica mostrata nella formula (50).

(50)		`\X y.X (\x.chase (y, x))`

Se (50) viene applicata a (43) il risultato dopo la riduzione di β è equivalente a (47) che è quello che si desiderava ottenere:

```
>>> lp = nltk.LogicParser()
>>> tvp = lp.parse(r'\X x.X(\y.chase(x,y))')
>>> np = lp.parse(r'(\P.exists x.(dog(x) & P(x)))')
```

```
>>> vp = nltk.sem.ApplicationExpression(tvp, np)
>>> print vp
(\X x.X(\y.chase(x,y)))(\P.exists x.(dog(x) & P(x)))
>>> print vp.simplify()
\x.exists z2.(dog(z2) & chase(x,z2))
```

Al fine di costruire una rappresentazione semantica per una frase, abbiamo anche bisogno di combinare NP nella semantica del soggetto. Se quest'ultimo è un'espressione quantificata come *every girl*, tutto procede allo stesso modo, come abbiamo mostrato in precedenza per *a dog barks*; il soggetto è tradotto con un'espressione di funzione che viene applicata alla rappresentazione semantica del VP. Tuttavia, ora sembra di aver creato un problema con i nomi propri. Finora, questi sono stati trattati semanticamente come costanti individuali, e quest'ultime non possono essere applicate come funzioni per le espressioni del tipo (47). Di conseguenza, per loro si necessita una rappresentazione semantica differente. In questo caso ciò che facciamo è re-interpretare i nomi propri, affinché anch'essi siano espressioni di funzione, come quantificatori NP. Qui di seguito troviamo l'espressione di λ richiesta per *Angus*.

(51)		\P.P(Angus)

(51) denota la funzione caratteristica corrispondente all'insieme di tutte le proprietà di Angus che sono vere. Convertire da una costante individuale angus a \P.P(angus) è un altro esempio di raccolta di tipo (type-raising), accennata in precedenza, e ci permette di sostituire un'applicazione con valori booleani come \x.walk(x)(angus) con un'applicazione di funzione equivalente \P.P(angus)(\x.walk(x)). Dalla riduzione di B, entrambe le espressioni riconducono a walk(angus) .

La grammatica simple-sem.fcfg contiene un piccolo insieme di regole per analisi e conversione di esempi semplici simili a quelli che abbiamo preso in esame. Eccone, di seguito, un po' più complicato.

```
>>> from nltk import load_parser
>>> parser = load_parser('grammars/book_grammars/simple-sem.fcfg',
trace=0)
>>> sentence = 'Angus gives a bone to every dog'
>>> tokens = sentence.split()
>>> trees = parser.nbest_parse(tokens)
```

```
>>> for tree in trees:
...     print tree.node['SEM']
all z2.(dog(z2) -> exists z1.(bone(z1) & give(angus,z1,z2)))
```

NLTK fornisce alcuni programmi di utilità per derivare ed ispezionare le interpretazioni semantiche. La funzione batch_interpret() è destinata all'interpretazione di un elenco di frasi input. Costruisce un dizionario d dove per ogni frase inviata nell'input, d [sent] è un elenco di coppie (*synrep*, *semrep*) costituito da diagrammi e rappresentazioni semantiche per sent. Il risultato è un elenco dal quale sent può risultare sintatticamente ambiguo; Nell'esempio seguente, tuttavia, there is only one parse tree per sentence in the list..

```
>>> sents = ['Irene walks', 'Cyril bites an ankle']
>>> grammar_file = 'grammars/book_grammars/simple-sem.fcfg'
>>> for results in nltk.batch_interpret(sents, grammar_file):
...     for (synrep, semrep) in results:
...         print synrep
(S[SEM=<walk(irene)>]
  (NP[-LOC, NUM='sg', SEM=<\P.P(irene)>]
    (PropN[-LOC, NUM='sg', SEM=<\P.P(irene)>] Irene))
  (VP[NUM='sg', SEM=<\x.walk(x)>]
    (IV[NUM='sg', SEM=<\x.walk(x)>, TNS='pres'] walks)))
(S[SEM=<exists z3.(ankle(z3) & bite(cyril,z3))>]
  (NP[-LOC, NUM='sg', SEM=<\P.P(cyril)>]
    (PropN[-LOC, NUM='sg', SEM=<\P.P(cyril)>] Cyril))
  (VP[NUM='sg', SEM=<\x.exists z3.(ankle(z3) & bite(x,z3))>]
    (TV[NUM='sg', SEM=<\X x.X(\y.bite(x,y))>, TNS='pres'] bites)
    (NP[NUM='sg', SEM=<\Q.exists x.(ankle(x) & Q(x))>]
      (Det[NUM='sg', SEM=<\P Q.exists x.(P(x) & Q(x))>] an)
      (Nom[NUM='sg', SEM=<\x.ankle(x)>]
        (N[NUM='sg', SEM=<\x.ankle(x)>] ankle)))))
```

Abbiamo visto ora come convertire le frasi inglesi in forme logiche e, prima, abbiamo visto come le forme logiche potevano essere verificato vere o false in un modello. Unendo queste due mappature, possiamo controllare il valore di verità delle frasi inglesi in un determinato modello. Consideriamo il modello m come sopra definito. L'utilità batch_evaluate() somiglia a batch_interpret() , fatta eccezione per il fatto che abbiamo bisogno di passare a un modello e all'assegnazione di una variabile come parametri. L'output è una tripla (*synrep*, *semrep*, *valore*) dove *synrep*, *semrep* sono come prima, e *value* è un valore di verità. Per semplicità, nell'esempio elabora solo una sola frase.

```
>>> v = """
... bertie => b
... olive => o
... cyril => c
... boy => {b}
... girl => {o}
... dog => {c}
... walk => {o, c}
... see => {(b, o), (c, b), (o, c)}
... """
>>> val = nltk.parse_valuation(v)
>>> g = nltk.Assignment(val.domain)
>>> m = nltk.Model(val.domain, val)
>>> sent = 'Cyril sees every boy'
>>> grammar_file = 'grammars/book_grammars/simple-sem.fcfg'
>>> results = nltk.batch_evaluate([sent], grammar_file, m, g)[0]
>>> for (syntree, semrep, value) in results:
...     print semrep
...     print value
all z4.(boy(z4) -> see(cyril,z4))
True
```

Il Quantificatore di ambiguità rivisitato

Un'importante limitazione dei metodi descritti sopra è che essi non si fronteggiano con la portata di ambiguità. Il nostro metodo di traduzione è basato sulla sintassi, nel senso che la rappresentazione

semantica è strettamente collegata all'analisi sintattica e la portata dei quantificatori nella semantica pertanto riflette la relativa portata del corrispondente s NP nella struttura di analisi sintattica. Di conseguenza, una frase come (26) sarà sempre tradotta come (53 a) e non (53 b)

(52)		Every girl chases a dog

(53)

a. `all x.(girl(x) -> exists y.(dog(y) & chase(x,y)))`

b. `exists y.(dog(y) & all x.(girl(x) -> chase(x,y)))`

Ci sono numerosi approcci per far fronte alla portata di ambiguità, e a breve guarderemo uno dei più semplici. Per cominciare, consideriamo la struttura delle formule di portata. La figura 10.3 rappresenta il modo in cui si differenziano le due letture di (52).

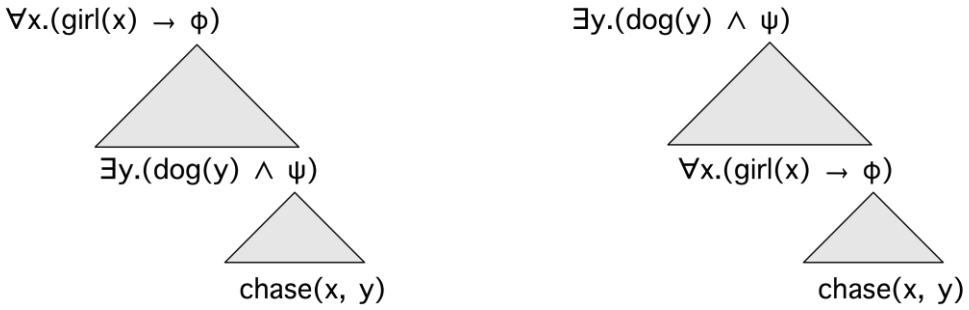


Figura 10.3: quantifier scopings

Consideriamo in primo luogo il lato sinistro della struttura. Nella parte superiore, abbiamo il quantificatore corrispondente a *every girl*. L'elemento ϕ può pensarsi come segnaposto per tutto ciò che si trova nell'ambito del quantificatore. Muovendoci verso il basso, notiamo che si può collegare il quantificatore corrispondente a *a dog* alla creazione di un'istanza di ϕ. Il risultato è un nuovo segnaposto ψ, che rappresenta la portata di *a dog*, e in questo noi possiamo collegare il 'core' della semantica, vale a dire la

frase aperta corrispondente a *x chases y*. La struttura sul lato destro è identica, tranne per il fatto che è stato ribaltato l'ordine dei quantificatori.

Nel metodo conosciuto come **archiviazione di Cooper** , una rappresentazione semantica non è più un'espressione della logica di primo ordine, ma è, al contrario, una coppia composta da una rappresentazione semantica "di base" e da un elenco di **operatori di associazione**. Per il momento, si pensi a un operatore di associazione come qualcosa di identico alla rappresentazione semantica di un quantificatore NP come in (44) o (45). Proseguendo lungo le linee indicate nella figura 10.3 supponiamo di aver costruito una rappresentazione semantica di una frase secondo lo stile di archiviazione di Cooper (52) e rendiamo il nostro core la formula aperta chase(x,y). Data una lista di operatori di associazione corrispondente ai due NP (52) scegliamo un operatore associazione fuori dalla lista e lo combiniamo con il core.

```
\P.exists y.(dog(y) & P(y))(\z2.chase(z1,z2))
```

In seguito prendiamo il risultato e applichiamo a questo il successivo operatore di associazione nella lista.

```
\P.all x.(girl(x) -> P(x))(\z1.exists x.(dog(x) & chase(z1,x)))
```

Una volta che l'elenco è vuoto, possediamo una forma convenzionale logica per la frase. La combinazione in questo modo degli operatori associazione con il core è chiamata **S-Retrieval (recupero di S)**. Se poniamo attenzione nel consentire ogni possibile ordine degli operatori di associazione (ad esempio, adottando tutte le permutazioni della lista), allora saremo in grado di generare qualsiasi tipo di ordine della portata dei quantificatori.

La prossima domanda è come costruire una rappresentazione composizionale core+store (nucleo + archivio). Come prima, ogni regola frasale e lessicale in grammatica avrà una caratteristica di sem , ma ora ci saranno incorporate le caratteristiche core e store. Per illustrare il meccanismo, consideriamo un esempio più semplice, vale a dire *Cyril smiles*. Ecco una regola lessicale per il verbo *smiles*(tratto dalla grammatica `storage.fcfg`) che sembra abbastanza innocuo.

```
IV[SEM=[core=<\x.smile(x)>, store=(/)]] -> 'smiles'
```

La regola per il nome proprio *Cyril* è più complessa.

```
NP[SEM=[core=<@x>, store=(<bo(\P.P(cyril),@x)>)]] -> 'Cyril'
```

Il predicato bo è composto da due parti: la rappresentazione standard di un nome proprio (la generazione del tipo) e l'espressione `@ x`, che si chiama **indirizzo** dell'operatore vincolante. `@ x` è una metavariable,

cioè, una variabile che va oltre le singole variabili della logica e, come si vedrà, fornisce anche il valore del core. La regola per VP filtra solo fino la semantica di IV e il lavoro interessante è svolto dalla regola di S.

```
VP[SEM=?s] -> IV[SEM=?s]


S[SEM=[core=<?vp(?subj)>, store=(?b1+?b2)]] ->
   NP[SEM=[core=?subj, store=?b1]] VP[SEM=[core=?vp, store=?b2]]
```

Il valore del core nel nodo s è il risultato dell'applicazione del valore core di VP, vale a dire `\x.smile(x)`, al valore del soggetto NP. Quest'ultimo non sarà `@ x`, ma piuttosto una creazione di istanza di `@ x`, detta `z3`. Dopo la riduzione di β, `<? vp(?subj) >` sarà unificato con `<smile(z3)>`. Ora, quando viene creata un'istanza di `@ x` come parte del processo di analisi, essa sarà essere istanziata uniformemente. In particolare, l'occorrenza di `@ x` nel soggetto NP's store (archivio di NP) verrà associata anche a `z3`, producendo l'elemento `bo(\P.P(cyril),z3)`. Questi passaggi sono visibili nella seguente struttura d'analisi.

```
(S[SEM=[core=<smile(z3)>, store=(bo(\P.P(cyril),z3))]]
  (NP[SEM=[core=<z3>, store=(bo(\P.P(cyril),z3))]] Cyril)
  (VP[SEM=[core=<\x.smile(x)>, store=()]]
    (IV[SEM=[core=<\x.smile(x)>, store=()]] smiles)))
```

Torniamo al nostro esempio più complesso,(52), e vediamo qual è il valore dello stile di archiviazione sem valore, dopo l'analisi con la grammatica `storage.fcfg` .

```
core  = <chase(z1,z2)>

store = (bo(\P.all x.(girl(x) -> P(x)),z1), bo(\P.exists x.(dog(x) &
P(x)),z2))
```

Ora dovrebbe essere più chiaro perché le variabili di indirizzo sono una parte importante dell'operatore vincolante. Ricordiamo che durante il recupero di S, si utilizzeranno operatori di associazioni fuori dalla lista prendendo gli operatori associazione fuori dalla lista `store` e per applicarli successivamente al core. Supponiamo di cominciare con `bo (\P.all x.(girl(x) - > P(x)),z1`), che vogliamo unire con `chase(z1,z2)`. La parte quantificante dell'operatore di associazione è `\P.all x.(girl(x) - > P(x))`, uniamola con `chase(z1,z2)`, che deve prima essere trasformato in un λ-abstract. Come facciamo a sapere su quale variabile applicare l'abstract? Questo è ciò che l'indirizzo z1 ci dice; vale a dire che *every girl* ha il ruolo di inseguitore piuttosto che quello di inseguimento.

Il modulo nltk.sem.cooper_storage si integra con il compito di trasformare le rappresentazioni semantiche di stile di archiviazione in forme logiche standard. In primo luogo, costruiamo un'istanza di CooperStore e ispezionare il suo store e il suo core .

```
>>> from nltk.sem import cooper_storage as cs

>>> sentence = 'every girl chases a dog'

>>> trees = cs.parse_with_bindops(sentence,
grammar='grammars/book_grammars/storage.fcfg')

>>> semrep = trees[0].node['SEM']

>>> cs_semrep = cs.CooperStore(semrep)

>>> print cs_semrep.core

chase(z1,z2)

>>> for bo in cs_semrep.store:

...     print bo

bo(\P.all x.(girl(x) -> P(x)),z1)

bo(\P.exists x.(dog(x) & P(x)),z2)
```

Infine chiamiamo `s_retrieve()` e controlliamo le letture.

```
>>> cs_semrep.s_retrieve(trace=True)

Permutation 1

   (\P.all x.(girl(x) -> P(x)))(\z1.chase(z1,z2))

   (\P.exists x.(dog(x) & P(x)))(\z2.all x.(girl(x) -> chase(x,z2)))

Permutation 2

   (\P.exists x.(dog(x) & P(x)))(\z2.chase(z1,z2))

   (\P.all x.(girl(x) -> P(x)))(\z1.exists x.(dog(x) & chase(z1,x)))
```

```
>>> for reading in cs_semrep.readings:

...     print reading

exists x.(dog(x) & all z3.(girl(z3) -> chase(z3,x)))

all x.(girl(x) -> exists z4.(dog(z4) & chase(x,z4)))
```

10.5 La semantica del discorso

Un **discorso** è una sequenza di frasi. Molto spesso, l'interpretazione di una frase in un discorso dipende da ciò che l'ha preceduta. Un chiaro esempio di questo proviene dai pronomi anaforici come *lui*, *lei* e ciò . Dato il discorso *Angus used to have a dog. But he recently disappeared.*, probabilmente interpreterai *he* come riferito al cane di Angus. Tuttavia, in *Angus used to have dog. He took him for walks in New Town,* ci sono maggiori probabilità di interpretare *he* come riferimento ad Angus stesso.

Teoria della rappresentazione del discorso

L'approccio standard alla quantificazione nella logica di primo ordine è limitato a singole frasi. Eppure sembrano esserci esempi dove la portata di un quantificatore può estendersi su due o più frasi. Abbiamo visto un esempio sopra, ed eccone, insieme a una traduzione.

(54)

a. Angus owns a dog. It bit Irene.

b. $\exists x.(dog(x) \wedge own(Angus, x) \wedge bite(x, Irene))$

Così, NP *a dog* si comporta come un quantificatore che si lega all' *it* nella seconda frase. La teoria della rappresentazione del discorso (Discourse Representation Theory, DRT) è stata sviluppato con l'obiettivo specifico di fornire un mezzo per la gestione di questo e altri fenomeni semantici che sembrano essere la caratteristica del discorso. Una **struttura di rappresentazione del discorso** (Discourse Representatione Structure, DRS) presenta il significato del discorso in termini di una lista dei referenti di discorso e un elenco delle condizioni. I **referenti del discorso** sono le cose in discussione nel discorso, ed essi corrispondono alle singole variabili della logica di primo ordine. Le condizioni della DRS si applicano a tali referenti e corrispondono alle formule atomiche aperte della logica di primo ordine. La figura 10.4 illustra come la DRS per la prima frase in (54 a) è aumentata per diventare una DRS per le due frasi.

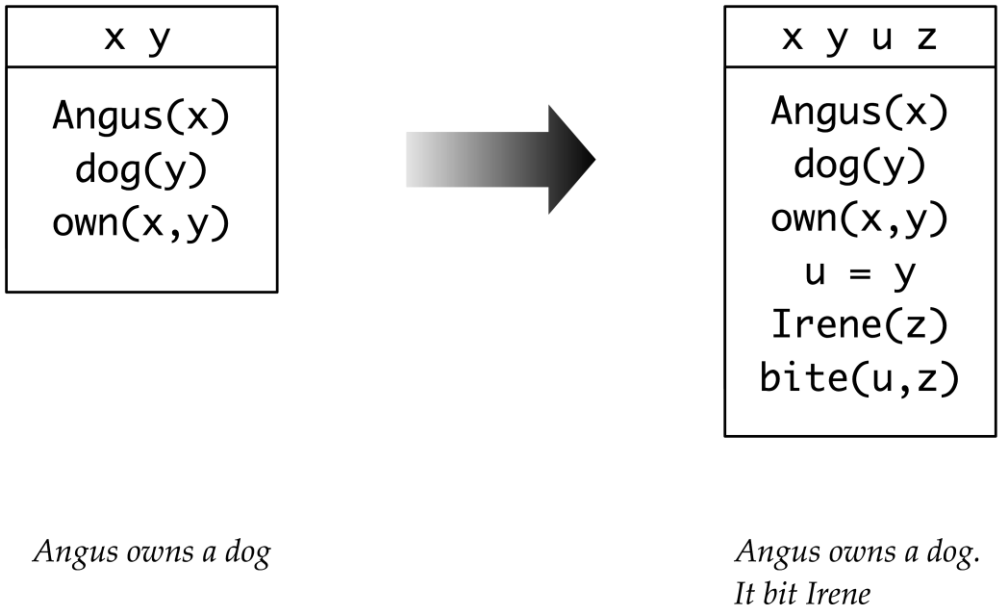


Figura 10.4: costruire una DRS; ls DRS sul lato sinistro rappresenta il risultato dell'elaborazione del discorso, la prima frase, mentre la DRS sulla destra mostra l'effetto della seconda frase di elaborazione e l'integrando del suo contenuto.

Quando viene elaborata la seconda frase di (54 a) essa viene interpretata nel contesto di ciò che è già presente nel lato sinistro della figura 10.4. Il pronome fa scattare l'aggiunta di un nuovo referente del discorso, diciamo *u*, e abbiamo bisogno di trovare un **precedente anaforico** per esso — cioè, si vuole agire al di fuori di ciò a cui questo si riferisce. Nella DRT, il compito di trovare l'antecedente per un pronome anaforico suppone il collegamento di questo a un referente del discorso già incluso nella corrente DRS, e *y* è la scelta più ovvia. Questo passaggio di elaborazione dà origine alla nuova condizione per cui *u* = *y*. Il resto del contenuto a cui ha contribuito la seconda frase si ricollega anche al contenuto della prima, e ciò è indicato sul lato destro del figura 10.4.

La figura illustra come una DRS può rappresentare più di una sola frase. In questo caso, è un discorso di due frasi, ma in linea di principio una singola DRS può corrispondere all'interpretazione di un testo intero. Possiamo indicare le condizioni di verità del lato destro della DRS nella figura 10.4. Informalmente, sarà vero in qualche situazione *s* se ci sono entità **a**, **c** e **i**, in *s*, corrispondenti ai referenti del discorso nella DRS tale che tutte le condizioni sono vere in *s* ; ovvero, **a** è chiamato *Angus*, **c** è un cane, **a** possiede **c**, **i** si chiama I**rene e c** ha morso **i**.

Al fine di elaborare le DRS computazionalmente, abbiamo bisogno di convertirle in un formato lineare. Ecco un esempio, nel quale la DRS è una coppia composta dall'elenco del discorso dei referenti e un elenco delle condizioni della DRS:

```
([x, y], [angus(x), dog(y), own(x,y)])
```

Il modo più semplice per creare un oggetto DRS in NLTK è mediante l'analisi di una rappresentazione di stringa.

```
>>> dp = nltk.DrtParser()
>>> drs1 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)])')
>>> print drs1
([x,y],[angus(x), dog(y), own(x,y)])
```

Possiamo usare il metodo Draw () per visualizzare il risultato, come illustrato nella figura 10.5

```
>>> drs1.draw()
```

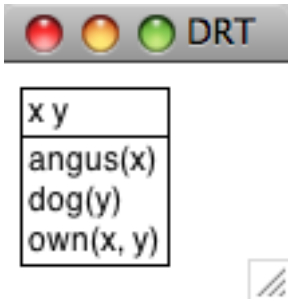


Figura 10.5: Schermata di una DRS

Quando abbiamo discusso le condizioni di verità delle DRS nella figura 10.4, abbiamo ipotizzato che i referenti di discorso in primo piano sono stati interpretati come quantificatori esistenziali, mentre le condizioni sono state interpretate come se fossero congiunte. In realtà, ogni DRS può essere tradotta in una formula della logica di primo ordine, e il metodo `fol()` implementa questa traduzione.

```
>>> print drs1.fol()
exists x y.((angus(x) & dog(y)) & own(x,y))
```

Oltre alle funzionalità disponibili per le espressioni della logica di primo ordine, le espressioni della DRT possiedono un operatore di concatenazione della DRS, rappresentato con il simbolo +. La concatenazione di due DRS è un singolo DRS contenente i referenti derivanti dalla fusione del discorso e le condizioni da entrambi gli argomenti. La concatenazione della DRS, convertita automaticamente in α, associa le variabili per evitare scontri di nome.

```
>>> drs2 = dp.parse('([x], [walk(x)]) + ([y], [run(y)])')
>>> print drs2
(([x],[walk(x)]) + ([y],[run(y)]))
>>> print drs2.simplify()
([x,y],[walk(x), run(y)])
```

Mentre tutte le condizioni viste finora sono state atomiche, è possibile incorporare una DRS all'interno di un'altra, e questo è il modo in cui la quantificazione è gestita universalmente. Nella `drs3`, non c'è nessun referente del discorso di primo livello, e la sola condizione è costituita da due sotto strutture, collegate da un'implicazione. Ancora una volta, possiamo utilizzare `fol()` per ottenere un handle sulle condizioni di verità.

```
>>> drs3 = dp.parse('([], [(([x], [dog(x)]) -> ([y],[ankle(y), bite(x,
y)]))])')
>>> print drs3.fol()
all x.(dog(x) -> exists y.(ankle(y) & bite(x,y)))
```

Abbiamo sottolineato all'inizio che la DRT è progettata per consentire l'interpretazione di pronomi anaforici attraverso il collegamento a referenti del discorso esistenti. La DRT impone vincoli nei quali i referenti del discorso sono "accessibili" come possibili antecedenti, ma non spiega il modo in cui un particolare antecedente è scelto tra una serie di candidati. Il modulo nltk.sem.drt_resolve_anaphora adotta allo stesso modo una strategia conservativa: se la DRS contiene una condizione di forma PRO(x), il metodo `resolve_anaphora()` lo sostituisce con una condizione di forma x = [...], dove [...] è un elenco di possibili antecedenti.

```
>>> drs4 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)])')
>>> drs5 = dp.parse('([u, z], [PRO(u), irene(z), bite(u, z)])')
>>> drs6 = drs4 + drs5
>>> print drs6.simplify()
([x,y,u,z],[angus(x), dog(y), own(x,y), PRO(u), irene(z), bite(u,z)])
>>> print drs6.simplify().resolve_anaphora()
([x,y,u,z],[angus(x), dog(y), own(x,y), (u = [x,y,z]), irene(z),
bite(u,z)])
```

Poiché l'algoritmo per la risoluzione dell'anafora è stato separato in un proprio modulo, ciò facilita lo scambio nei procedimenti alternativi che cercano di rendere più intuitive le ipotesi per quanto riguarda l'antecedente corretto.

Il nostro trattamento delle DRS è completamente compatibile con i meccanismi esistenti per la gestione dell'astrazione di λ, e di conseguenza è semplice per la creazione di rappresentazioni semantiche compositive che si basano sulla DRT piuttosto che sulla logica di primo ordine. Questa tecnica è illustrata nella seguente regola per gli indefiniti (parte della grammatica drt.fcfg). Per una facilità di paragone, abbiamo aggiunto la regola parallela per gli indefiniti da `semplice-sem.fcfg`.

```
Det[num=sg,SEM=<\P Q.(([x],[]) + P(x) + Q(x))>] -> 'a'

Det[num=sg,SEM=<\P Q. exists x.(P(x) & Q(x))>] -> 'a'
```

Per avere un'idea più chiara di come funziona la regola della DRT, si guardi questa sottostruttura per NP *a dog*.

```
(NP[num='sg', SEM=<\Q.(([x],[dog(x)]) + Q(x))>]
  (Det[num='sg', SEM=<\P Q.((([x],[]) + P(x)) + Q(x))>] a)
  (Nom[num='sg', SEM=<\x.([],[dog(x)])>]
    (N[num='sg', SEM=<\x.([],[dog(x)])>] dog))))
```

L'astrazione di λ per l'indefinito viene applicato come un'espressione di funzione `\x.([],[dog(x)])` che porta a `\Q.(([x],[]) + ([in], [dog(x)]) + Q(x))`; Dopo la semplificazione, otteniamo come rappresentazione per il complesso di NP, `\Q.(([x],[dog(x)]) + Q(x))`.

Al fine di compiere un'analisi con la grammatica `drt.fcfg`, specifichiamo nella chiamata a `load_parser()` che i valori di SEM nelle strutture caratteristiche devono essere analizzati utilizzando `DrtParser` al posto del predefinito LogicParser .

```
>>> from nltk import load_parser

>>> parser = load_parser('grammars/book_grammars/drt.fcfg',
logic_parser=nltk.DrtParser())

>>> trees = parser.nbest_parse('Angus owns a dog'.split())

>>> print trees[0].node['SEM'].simplify()

([x,z2],[Angus(x), dog(z2), own(x,z2)])
```

Elaborazione del discorso

Quando interpretiamo una frase, utilizziamo un ricco contesto, determinato in parte dal contesto precedente e in parte dalle nostre ipotesi di partenza. La DRT fornisce la teoria di come il significato di una frase sia integrato in una rappresentazione del discorso preliminare. Due cose, però, non sono state affrontate nell'approccio di elaborazione appena discusso. In primo luogo, non c'è stato nessun tentativo di incorporare qualsiasi tipo di inferenza; in secondo luogo, sono state elaborate solo singole frasi. Queste omissioni vengono sanate dal modulo `nltk.inference.discourse` .

Considerando che un discorso è una sequenza s_1,... s_n di frasi, un *filo del discorso* è una sequenza s_1- r_{mi},... s_n-r_j di letture, una per ogni frase del discorso. Il modulo elabora le frasi in modo incrementale tenendo traccia di tutte le possibili discussioni nel caso in cui ci fosse ambiguità. Per semplicità, nell'esempio è ignorata la portata di ambiguità.

```
>>> dt = nltk.DiscourseTester(['A student dances', 'Every student is a
person'])

>>> dt.readings()
```

S0 letture:

r0 s0: esiste x.(student(x) & dance(x))

S1 letture:

S1-r0: tutti i x.(student(x) - > person(x))

Quando una nuova frase viene aggiunta al discorso corrente, impostando il parametro `consistchk = True`, avvia un controllo richiamando il modello controllore per ogni thread, vale a dire, una sequenza di letture ammissibili. In questo caso, l'utente ha la possibilità di ritrattare la frase in questione.

```
>>> dt.add_sentence('No person dances', consistchk=True)

Inconsistent discourse d0 ['s0-r0', 's1-r0', 's2-r0']:

  s0-r0: exists x.(student(x) & dance(x))

  s1-r0: all x.(student(x) -> person(x))

  s2-r0: -exists x.(person(x) & dance(x))

>>> dt.retract_sentence('No person dances', verbose=True)

Current sentences are

s0: A student dances

s1: Every student is a person
```

In maniera simile, usiamo `informchk = True` per verificare se una nuova frase ϕ è informativa in relazione al discorso corrente. La Theorem prover considera le frasi esistenti nel thread come ipotesi e tenta di dimostrare ϕ; essa è informativa se non viene trovata alcuna prova.

```
>>> dt.add_sentence('A person dances', informchk=True)

Sentence 'A person dances' under reading 'exists x.(person(x) &
dance(x))':

Not informative relative to thread 'd0'
```

È anche possibile passare ad una serie supplementare di ipotesi come la conoscenza di partenza ed utilizzare queste per filtrare le nostre letture incoerenti; si veda il discorso HOWTO a http://www.nltk.org/howto per maggiori dettagli.

Il modulo di discorso può ospitare l'ambiguità semantica e filtrare le letture che non sono ammissibili. Nell'esempio seguente viene richiamata sia la Semantica di Glue, nonché la DRT. Poiché il modulo della semantica di Glue è configurato per utilizzare il parser di ampia copertura Malt, l'input (*Every dog chases a boy. He runs*.) needs to be tagged as well as tokenized.

```
>>> from nltk.tag import RegexpTagger

>>> tagger = RegexpTagger(

...     [('^(chases|runs)$', 'VB'),

...      ('^(a)$', 'ex_quant'),

...      ('^(every)$', 'univ_quant'),

...      ('^(dog|boy)$', 'NN'),

...      ('^(He)$', 'PRP')

... ])

>>> rc =
nltk.DrtGlueReadingCommand(depparser=nltk.MaltParser(tagger=tagger))

>>> dt = nltk.DiscourseTester(['Every dog chases a boy', 'He runs'],
rc)

>>> dt.readings()
```

S0 letture:

r0 s0: ([in], [(([x],[dog(x)]) - > ([z3],[boy(z3), chases(x,z3)]))]) s0-r1: ([z4],[boy(z4), (([x],[dog(x)]) - > ([], [chases(x,z4)]))])

S1 letture:

S1-r0: ([x],[PRO(x), runs(x)])

La prima frase del discorso ha due possibili letture, a seconda dell'ambito del quantificatore. La lettura unica della seconda frase rappresenta il pronome *he* tramite la condizione *PRO(x)'*. Diamo un'occhiata ai fili (threads) del discorso che risultano:

```
>>> dt.readings(show_thread_readings=True)
d0: ['s0-r0', 's1-r0'] : INVALID: AnaphoraResolutionException
d1: ['s0-r1', 's1-r0'] : ([z6,z10],[boy(z6), (([x],[dog(x)]) ->
([],[chases(x,z6)])), (z10 = z6), runs(z10)])
```

Quando si esaminano i thread d0 e d1, vediamo che leggendo s0 r0, dove *every dogs* fuori dall'ambito a boy, è ritenuto inammissibile perché il pronome nella seconda frase non può essere risolto. Al contrario, nel thread d1 il pronome (rinominato z24) è stata associato *tramite* l'equazione (z24 = z20) .

Le letture inammissibili possono essere filtrate attraverso il parametro `filtro = True`.

```
>>> dt.readings(show_thread_readings=True, filter=True)
d1: ['s0-r1', 's1-r0'] : ([z12,z15],[boy(z12), (([x],[dog(x)]) ->
([],[chases(x,z12)])), (z17 = z12), runs(z15)])
```

Anche se questo breve discorso è estremamente limitato, dovrebbe dare l'idea del tipo dei problemi di elaborazione semantica che sorgono quando si va al di là delle singole frasi, andiamo di là di singole frasi e anche un'idea di quali siano le tecniche che possono essere utilizzate per affrontare tali problemi.

10.6 Sommario

- La logica di primo ordine è un linguaggio adatto per la rappresentazione del senso di un linguaggio naturale in una cornice computazionale, poiché è abbastanza flessibile per rappresentare molti aspetti utili del significato naturale, e ci sono efficienti THEOREM PROVERS per ragionare con la logica di primo ordine. (Allo stesso modo, ci sono una varietà di fenomeni nella semantica del linguaggio naturale che si crede richiedano meccanismi logici più potenti.)

- Oltre alla traduzione di frasi in linguaggio naturale nella logica di primo ordine, possiamo affermare le condizioni di verità di queste frasi esaminando i modelli delle formule di prime ordine.
- Al fine di costruire composizionalmente le rappresentazioni di significato, abbiamo completato la logica del primo ordine con il calcolo di λ.
- La riduzione di B nel calcolo λ-calcolo infinitesimale- corrisponde semanticamente all'applicazione di una funzione su un argomento. Sintatticamente, consiste nella sostituzione di una variabile vincolata da λ nell'espressione di funzione, con l'espressione che fornisce l'argomento nell'applicazione di funzione.
- Una parte fondamentale della costruzione di un modello si trova nella costruzione di una valutazione che assegna le interpretazioni a costanti non logiche. Queste vengono interpretate sia come s*n*-ary predicati o come singole costanti.
- Un'espressione aperta è un'espressione contenente una o più variabili libere. Le espressioni aperte ricevono un'interpretazione solo quando le loro variabili libere ricevono i valori dall'assegnazione di una variabile.
- I quantificatori vengono interpretati dalla costruzione, per una formula ϕ [*x*] aperta nella variabile *x*, dell'insieme di individui che rendono ϕ [*x*] vera quando un'assegnazione *g* li assegna come il valore di *x*. Il quantificatore poi pone vincoli su quell'insieme.
- Un'espressione chiusa è un'espressione che non ha nessuna variabile libera; cioè, le variabili sono tutte associate. Una frase chiusa è vera o falsa rispetto a tutte le assegnazioni di variabili.
- Se due formule si differenziano solo per l'etichetta della variabile vincolata dall'operatore vincolante (vale a dire, λ o un quantificatore), si dicono equivalenti di α. Il risultato di una variabile associata in una formula di rietichettatura viene chiamato conversione di α.
- Data una formula con due quantificatori nidificati Q_1 e Q_2, si dice che il quantificatore ultraperiferico Q_1 disponga di ampia portata (o portata superiore a Q_2). Le frasi in inglese sono spesso ambigue rispetto all'ambito dei quantificatori che esse contengono.
- Le frasi in inglese possono essere associate a una rappresentazione semantica trattando sem come una caratteristica all'interno di una grammatica basata su caratteristiche. Il valore di espressioni complesse sem tipicamente coinvolge l'applicazione funzionale del valore sem delle espressioni componenti.

10.7 Ulteriori letture

Per ulteriori materiali su questo capitolo e su come installare il Prover9 theorem prover e Mace4 modello builder, consultare http://www.nltk.org/ . Altre informazioni genreali sono date da McCune, 2008.

Per ulteriori esempi di analisi semantica con NLTK, vedere la semantica e la logica HOWTO presso www.nltk.org/howto. Notare che ci sono le implementazioni di altri due approcci sulla portata di ambiguità, vale a dire **la Hole Semantics** come descritta in Blackburn & Bos, 2005. e la **Glue Semantics** come descritta in Dalrymple, 1999.

Ci sono molti fenomeni all'interno della semantica del linguaggio naturale che non sono stati toccati in questo capitolo. In particolare:

- il tempo e l'aspetto degli eventi;
- i ruoli semantici;
- i quantificatori generalizzati, come *la maggior parte dei*;
- costruzioni intenzionali che coinvolgono, ad esempio, i verbi come *può* e *credere*.

Mentre (1) e (2) possono essere affrontati utilizzando la logica di primo ordine, (3) e (4) richiedono logiche diverse.

Una panoramica completa dei risultati e delle tecniche nella costruzione di linguaggio naturale front-end per database può essere trovata in Androutsopoulos, Ritchie e Thanisch, 1995.

Qualsiasi libro introduttivo alla logica moderna presenterà la logica proposizionale e di primo ordine. Hodges, 1977, è altamente raccomandato come un testo divertente e penetrante.

Per un libro di testo ad ampio raggio, si consigliano due volumi sulla logica, che presentano anche materiale contemporaneo sulla semantica formale del linguaggio naturale, inclusa la grammatica di Montague e la logica intenzionale, si veda Gamut,1991. Kamp & Reyle, 1993, e fornisce il resoconto definitivo della Teoria della rappresentazione del discorso (DRT) e si occupa di un interessante parte del linguaggio naturale, includento il tempo, l'aspetto e la modalità. Un altro studio completo della semantica di molte costruzioni di linguaggio naturale è Carpenter, 1997.

Ci sono numerose opere che introducono la logica semantica nell'ambito della teoria linguistica. Chierchia & McConnel-Ginet, 1990, è relativamente agnostico sulla sintassi, mentre Heim & Kratzer, 1998, e Larsone & Segal, 1995, sono entrambi più esplicitamente orientati verso l'integrazione di una semantica condizionata dalla verità, in un quadro Chomskiano.

Blackburn & Bos, 2005, è il primo libro di testo dedicato alla semantica computazionale e fornisce un'eccellente introduzione alla tema. Approfondisce molti degli argomenti trattati in questo capitolo, tra cui underspecification del quantificatore della portata di ambiguità, l'inferenza di primo ordine e l'elaborazione del discorso.

Per ottenere una panoramica del più avanzato approccio alla semantica contemporaneo, si consultino Lappin, 1996, o Benthem e Meulen, 1997.