Anàlisi textual i processament de corpus

Antoni Oliver Gonzalez

PID_00155227

Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

En aquest mòdul aprendrem a fer un tractament molt bàsic de textos i corpus textuals. Aquests tractaments bàsics consistiran en el comptatge de paraules, segmentació en unitats lèxiques i segmentació del text en oracions. Aquests processos bàsics presenten una sèrie de problemes que convé resoldre satisfactòriament. En altres mòduls començarem a fer anàlisis lingüístiques (morfològica, sintàctica, etc.) però els passos que aprendrem a fer en aquest mòdul seran imprescindibles per a l'èxit de les anàlisis més profundes.

Per a poder tractar correctament arxius i corpus textuals cal conèixer a fons les diferents codificacions de caràcters. En aquest mòdul aprendrem a obrir fitxers en diferents codificacions i veure els avantatges que proporciona la codificació Unicode.

Els corpus lingüístics poden estar anotats amb diferents informacions lingüístiques. En aquest mòdul veurem alguns d'aquests nivells i com podem llegir corpus anotats amb l'NLTK.

Finalment també veurem alguns recursos lèxics proporcionats per l'NLTK i dedicarem una atenció especial al cas del WordNet.

Per a poder fer els exemples del mòdul és imprescindible tenir instal·lat Python i NLTK en el nostre ordinador. En l'annex 1 podeu trobar tots els detalls de com fer aquesta instal·lació.

Objectius

Els objectius bàsics que ha d'haver aconseguit l'estudiant una vegada treballats els continguts d'aquest mòdul són els següents:

1. Comprendre els conceptes d'ocurrència (*token*) i tipus (*type*).
2. Saber processar textos i corpus amb NLTK.
3. Saber accedir a textos disponibles en Internet.
4. Conèixer les diferents codificacions de caràcters, com processar-les i transformar textos entre diferents codificacions.
5. Comprendre quins avantatges ofereix Unicode per a la representació de caràcters.
6. Saber en què consisteix la tasca de segmentació en unitats lèxiques (*tokenització*), les dificultats que presenta i les principals tècniques per a portar-la a terme.
7. Saber en què consisteix la tasca de segmentació del text, les dificultats que presenta i les principals tècniques per a portar-la a terme.
8. Saber calcular freqüències absolutes, relatives i distribucions de freqüència condicionals sobre un corpus amb l'NLTK.
9. Conèixer els diferents tipus d'anotacions que pot presentar un corpus i com accedir a la informació dels corpus anotats amb l'NLTK.
10. Conèixer els recursos lèxics disponibles a l'NLTK i molt especialment el WordNet.

1. Tractament de fitxers: ocurrències (*tokens*) i tipus (*types*)

1.1. Obrir i llegir un fitxer de text

En aquest subapartat aprendrem a obrir un fitxer de text amb Python i llegir i processar-ne el contingut. Proveu el programa següent (`programa-3-1.py`):

```
f=open("noticia1.txt","rU")
text=f.read()
print text
```

En la primera línia obrim un fitxer, anomenat `noticia1.txt`. El paràmetre `rU` significa "de lectura" (r: *read*) i que ignori les diferents convencions per a indicar línia nova (U: *universal*). A la segona línia llegim tot el contingut del fitxer i el posem a la variable `text`. A la tercera línia escrivim per pantalla el valor de la variable `text`. Ara per ara, si els accents no es veuen bé, no importa, ja arreglarem aquests detalls més endavant.

1.2. Ocurrències (*tokens*) i tipus (*types*)

Podem modificar el programa del subapartat anterior perquè ens doni totes les paraules del text. El programa (`programa-3-2.py`) quedaria de la manera següent:

```
f=open("noticia1.txt","rU")
text=f.read()
paraules=text.split();
print paraules
totalparaules=len(paraules)
print "Total paraules: ", totalparaules
```

En la tercera línia estem separant les paraules del text (considerant que una paraula està delimitada per espais en blanc). A la quarta línia escrivim les paraules per pantalla, a la cinquena calculem el nombre de paraules del text i a la sisena escrivim el nombre total de paraules. El que hem calculat en aquest programa són les **ocurrències** (*tokens*) del text, i moltes d'aquestes paraules estaran repetides a la llista, ja que ocorren més d'una vegada. En aquest exemple segur que no veureu correctament els caràcters accentuats ni altres que no

coincideixin amb el llatí bàsic. No us preocupeu, és normal i explicarem els motius i com podem solucionar-ho més endavant en aquest mateix mòdul.

Amb el programa següent (`programa-3-3.py`) calcularem els **tipus** (*types*) que ocorren en aquest text, és a dir, les paraules diferents. Farem ús de la instrucció `set`, que crea una col·lecció d'elements únics, que no mantenen cap mena d'ordre.

```
f=open("noticia1.txt","rU")
text=f.read()
paraules=text.split();
tipus=set(paraules)
print tipus
totaltipus=len(tipus)
print "Total tipus: ", totaltipus
```

Podem resumir que les *ocurrències* són el nombre total de paraules que apareixen en el text i els *tipus* el nombre de paraules diferents que apareixen en el text.

De fet, no podem parlar estrictament de paraules, ja que també s'inclouen els signes de puntuació, les xifres, etc. Podem calcular un índex de riquesa lèxica dividint el nombre d'ocurrències entre el nombre de tipus (l'`import` de l'inici del `programa-3-3b.py` ens assegura que la divisió sigui en coma flotant):

```
from __future__ import division
f=open("noticia1.txt","rU")
text=f.read()
paraules=text.split();
totalparaules=len(paraules)
print "Total paraules: ", totalparaules
tipus=set(paraules)
totaltipus=len(tipus)
print "Total tipus: ", totaltipus
riquesalexica=len(paraules) / len(tipus)
print "Riquesa lexica: ", riquesalexica
```

1.3. Obrir i tractar tots els fitxers d'un directori

En els exemples dels subapartats anteriors hem obert un fitxer de text i hem fet un processament senzill: el càlcul d'ocurrències i de tipus. Quan tractem amb col·leccions grans de textos pot ser de gran utilitat tenir tots els fitxers en un directori i processar-los tots, un darrere de l'altre.

Python ens ofereix la possibilitat de llegir tots els fitxers d'un determinat directori. Fixem-nos en el programa següent (`programa-3-4.py`).

```
import os
directori="/home/aoliver/"
for f in os.listdir(directori):
    print f
```

Ajusteu el valor de la variable `directori` a un valor real del vostre ordinador i sistema operatiu. Aquest programa fa una llista del nom de tots els arxius del directori; si us hi fixeu dóna una llista només del nom dels arxius. Hi ha una altra possibilitat per a assolir el mateix (`programa-3-4b.py`).

```
import glob
directori="/home/aoliver/*"
for f in glob.glob(directori):
    print f
```

Fixeu-vos que en aquest cas el programa ens fa una llista tant del nom de l'arxiu com del directori.

Ara ens interessa obrir tots els fitxers de text (que tenen extensió `.txt`) i imprimir el seu contingut per pantalla; haurem d'afegir algunes instruccions (`programa-3-5.py`).

```
import os
directori="/home/aoliver"
for f in os.listdir(directori):
    if (os.path.splitext(f)[1]==".txt"):
        print f
        f=open(os.path.join(directori,f),"rU")
        text=f.read()
        print text
```

Recordeu que heu d'ajustar el valor de la variable `directori` a un directori real del vostre ordinador i sistema operatiu. `os.listdir(directori)` retorna una llista amb tots els fitxers del directori, que recorrem amb el bucle `for`. La instrucció `os.path.splitext(f)` divideix el nom i l'extensió de l'arxiu `f`, i en la posició 1 guarda l'extensió. Comprovem si és igual a `.txt` i si ho és obrim l'arxiu. Fixeu-vos que en `f` només tenim el nom de l'arxiu i que per a poder-lo obrir necessitem tenir també el directori on es troba. Això ho aconseguim amb `os.path.join(directori,f)`.

Aquest mateix programa es pot simplificar una mica (`programa-3-5b.py`) fent servir `glob`.

```
import glob
directori="/home/aoliver/*.txt"
for f in glob.glob(directori):
    print f
    f=open(f,"rU")
    text=f.read()
    print text
```

Exercici

Creeu un directori i poseu diversos arxius de text. Modifiqueu el `programa-3-5.py` o el `programa-3-5b.py` perquè calculi les ocurrències i els tipus de tots els arxius d'aquest directori.

1.4. Obrir i tractar tots els fitxers d'un directori de manera recursiva

En algunes ocasions ens pot interessar tractar tots els fitxers d'un determinat tipus d'un directori i de tots els subdirectoris que pengen d'aquest, és a dir, tractar tots els fitxers d'un directori de manera recursiva. Això pot ser d'utilitat per a organitzar els fitxers del nostre corpus en subcarpetes segons diversos criteris i poder accedir a tots els arxius.

```
import os

def dir_rec(path):
    result=[]
    temp=os.listdir(path)
    for a in temp:
        fp=path+"/"+a
        if (os.path.isfile(fp)):
            result.append(fp)
        elif (os.path.isdir(fp)):
            files=dir_rec(fp)
            result.extend(files)
    return(result)

arxius=dir_rec('/home/aoliver')
print arxius
for arxiu in arxius:
    print arxiu
```

En el programa anterior (`programa-3-6.py`) hem creat un funció anomenada `dir_rec` que retorna els noms dels arxius d'un directori i si el directori conté un subdirectori es crida ella mateixa per a trobar els arxius i possibles subdirectoris. És a dir, aquesta funció es crida recursivament si troba un subdirectori.

Recursivitat

Per a tractar els directoris de manera recursiva la funció `dir_rec` del nostre programa es crida ella mateixa quan troba un nou subdirectori.

1.5. Agafar text d'Internet

Internet pot ser una font pràcticament inesgotable de textos per a fer diferents estudis lingüístics. Hi ha hagut molts estudis sobre l'ús d'Internet com a corpus, i fins i tot la prestigiosa revista *Computational Linguistics* ha dedicat un número sencer a aquest tema (*Special Issue on the Web as a Corpus* (vol. 29, núm. 3, setembre 2003) (Kilgarriff i Grefenstette, 2003). Cal esmentar també el corpus Cuc Web (*http://ramsesii.upf.es/cgi-bin/cucweb/search-form.pl*) (Boleda i altres, 2004) que s'ha confeccionat descarregant tot el contingut d'Internet en català.

Ús indiscriminat d'Internet

Com ja sabem, a Internet hi ha de tot, i l'ús indiscriminat de textos d'Internet pot servir per a fer estudis sobre l'ús del llenguatge en aquest mitjà, però no per a altres estudis. Ara bé, si seleccionem bé el contingut que descarreguem (seleccionant una o més fonts que s'adaptin als nostres requisits), Internet serà un recurs molt valuós per a confeccionar els nostres corpus. En aquest subapartat aprendrem a descarregar i processar textos d'Internet.

Amb Python és molt senzill agafar directament el text d'una plana web (evidentment cal disposar d'una connexió a Internet). Proveu el programa següent (`programa-3-7.py`).

```
from urllib import urlopen
plana=urlopen("http://www.vilaweb.cat/www/noticia?p_idcmp=3625805").read()
print plana
```

Com podreu observar, si fem això visualitzarem el contingut de la plana amb els seus codis HTML. Caldrà buscar estratègies per a treure les marques HTML del text. Una possibilitat és fer servir expressions regulars com en el programa següent (`programa-3-8.py`).

```
from urllib import urlopen
import re
plana=urlopen("http://www.vilaweb.cat/www/noticia?p_idcmp=3625805").read()
planamod=re.sub(r'<[^>]*>','',plana);
print planamod
```

Com podeu observar, el filtratge no és perfecte, però la sortida ha millorar força. Quan aprofundim una mica en l'ús de les expressions regulars podrem tornar a aquest programa i acabar de polir-lo.

NLTK també proporciona un "netejador" de planes web que les transforma a text amb resultats similars (`programa-3-8b.py`).

```
import nltk
from urllib import urlopen
plana=urlopen("http://www.vilaweb.cat/www/noticia?p_idcmp=3625805").read()
text = nltk.clean_html(plana)
print text
```

Si la plana està mal formada el netejador llençarà un error i no obtindrem el resultat esperat.

Si volem confeccionar un corpus a partir d'Internet haurem d'anar llegint diverses planes i guardant el contingut en fitxers de text. Amb el programa següent (`programa-3-9.py`) descarreguem tot un any del diari *El Periódico de Catalunya*.

```
from urllib import urlopen

def substitucions(linia):
    liniamod=re.sub(r'</p>','\n',linia)
    liniamod=re.sub(r'<br />','\n',liniamod)
    liniamod=re.sub(r'<br/>','\n',liniamod)
    liniamod=re.sub(r'<[^>]*>','',liniamod)
    liniamod=re.sub(r'&#39;','\'',liniamod)
    liniamod=re.sub(r'&aacute;','á',liniamod)
    liniamod=re.sub(r'&eacute;','é',liniamod)
    liniamod=re.sub(r'&iacute;','í',liniamod)
    liniamod=re.sub(r'&oacute;','ó',liniamod)
    liniamod=re.sub(r'&uacute;','ú',liniamod)
    ...
    liniamod=re.sub(r'S"','S\'',liniamod)
    return(liniamod)

def baixa(enllac,nomarxiu):
    print enllac
    print nomarxiu
    fcat=open(nomarxiu,"w")
    text=urlopen(enllac).read()
    mo=re.findall("<p>(.+)</p>",text)
    for l in mo:
        l=substitucions(l)
        if re.match("[a-zA-z]",l):
            fcat.write(l+"\n")
    fcat.close()


def processa_data(any,data):
    url="http://www.elperiodico.cat/archivo_titulares.asp?f="+data
```

```
    text=urlopen(url).read()
    mo=re.findall(”<a href=\”(http://www.elperiodico.com/.+?idnoticia.+?)\”>”,text)
    expreg=re.compile(”http://www.elperiodico.com/default.asp\?idpublicacio\_PK\
                      =(.+?)&idioma\=CAT\&idnoticia\
    _PK\=(.+?)\&idseccio\_PK=([0-9]+)”);
    for m in mo:
        t = expreg.match(m)
        if t:
            idpublicacio=t.group(1)
            idnoticia=t.group(2)
            idseccio=t.group(3)
            nomarxiu=”./”+any+”/”+data+”-”+idpublicacio+”-”+idseccio+”-”
                     +idnoticia+”.cat”;
            baixa(m,nomarxiu)
            enllacesp=m.replace(”CAT”,”CAS”)
            nomarxiuesp=”./”+any+”/”+data+”-”+idpublicacio+”-”+idseccio+”-”
                     +idnoticia+”.spa”
            baixa(enllacesp,nomarxiuesp);

mes=(”01”,”02”,”03”,”04”,”05”,”06”,”07”,”08”,”09”,”10”,”11”,”12”)
dia=(”01”,”02”,”03”,”04”,”05”,”06”,”07”,”08”,”09”,”10”,”11”,”12”,”13”,
”14”,”15”,”16”,”17”,”18”,”19”,”20”,”21”,”22”,”23”,”24”,”25”,”26”,”27”,
”28”,”29”,”30”,”31”)
any=”2008”;
for m in mes:
    for d in dia:
        data=any+m+d
        print data
        processa_data(any,data)
```

Aquest programa està organitzat en una sèrie de funcions (`substitucions`, `baixa` i `processa_data`). El programa principal (la part de codi de baix de tot) és molt senzill. Simplement tenim una llista de mesos i de dies i la variable `any` fixada a 2008. Amb tot això i dos bucles niats va creant dates (algunes són incorrectes, com el 31 de febrer, però no importa perquè simplement la resta del programa no trobarà cap notícia amb aquesta data). Un cop creada la data crida la funció `processa_data`. Aquesta funció obre la plana de titulars del dia indicat per la data i cerca tots els enllaços que són notícies. Amb això crea un enllaç adequat i un nom d'arxiu i crida la funció `baixa`. Aquesta funció simplement descarrega la notícia, cerca totes les línies amb les marques de paràgraf, aplica la funció `substitucions` i guarda el contingut de cada notícia en un fitxer de text. La funció `substitucions` canvia les entitats HTML pels caràcters associats. La funció l'hem simplificada aquí perquè hi ha moltes entitats i en el fitxer està en versió completa.

Per a executar el programa cal crear una subcarpeta que s'anomeni `2008` just per sota d'on es troba el programa mateix. Per a poder fer aquest programa s'ha hagut d'analitzar com estan estructurats els arxius de la web del diari i s'ha pogut fer de manera que desi els arxius en castellà i en català amb el mateix nom i amb una extensió diferent.

Cal recordar que aquests continguts estan protegits per drets d'autor i que els fitxers descarregats no es podran distribuir.

En tot cas no cal que l'executeu sencer; quan hagi descarregat uns quants arxius podeu parar l'execució del programa i observar-ne els resultats.

1.6. Agafar text dels corpus de l'NLTK

L'NLTK es distribueix amb una gran quantitat de corpus. Tot i que una gran part d'aquests corpus estan en anglès, n'hi ha també en altres llengües, com el català i el castellà. Podem agafar el text d'un d'aquest corpus i fer diferents tipus de processos. Per exemple (`programa-3-10.py`):

```
from nltk.corpus import gutenberg
contingut=gutenberg.fileids()
print contingut
```

Aquest programa simplement mostra el contingut del corpus Gutenberg.

```
['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']
```

Podem fer moltes altres coses; per exemple en `programa-3-10b.py` agafem el *Hamlet* de Shakespeare i mirem les concordances de la paraula *better*.

```
import nltk
text=nltk.Text(nltk.corpus.gutenberg.words('shakespeare-hamlet.txt'))
text.concordance("better")
```

Fixeu-vos en la primera línia i com hem canviat la manera d'importar el corpus respecte al `programa-3-10.py`. En el primer cas hem importat únicament el corpus Gutenberg i en la segona hem importat tot l'NLTK. En el segon cas hem hagut d'especificar a la instrucció `nltk.corpus.gutenberg` i en el primer cas només `gutenberg`. Totes dues formes són equivalents tot i que és

una bona pràctica, per a fer programes més eficients, importar només allò que realment ens cal per al nostre programa.

La sortida d'aquest programa és:

```
Building index...
Displaying 15 of 15 matches:
; nor haue we heerein barr ' d Your better Wisedomes , which haue freely gone
ke sanctified and pious bonds , The better to beguile . This is for all : I wo
ade him mad . I am sorrie that with better speed and iudgement I had not quote
eparation ' gainst the Poleak : But better look ' d into , he truly found It w
d loue , and by what more deare , a better proposer could charge you withall ;
e both in reputation and profit was better both wayes Rosin . I thinke their I
s most like if their meanes are not better ) their Writers do them wrong , to
 time . After your death , you were better haue a bad Epitaph , then their ill
ir desart Ham . Gods bodykins man , better . Vse euerie man after his desart ,
Ophe . Could Beautie my Lord , haue better Comerce then your Honestie ? Ham .
se me of such things , that it were better my Mother had not borne me . I am v
 , to take off my edge Ophe . Still better and worse Ham . So you mistake Husb
rough our bad performance , ' Twere better not assaid ; therefore this Proiect
ue seene you both : But since he is better ' d , we haue therefore oddes Laer
e , The King shal drinke to Hamlets better breath , And in the Cup an vnion sh
```

L'accés als corpus de l'NLTK es fa mitjançant la classe `nltk.corpus.reader`. A la taula 1 podem observar els mètodes bàsics d'aquesta classe. Recordeu que podeu accedir a la documentació completa fent `help(nltk.corpus.reader)`.

Taula 1. Alguns mètodes de la classe `nltk.corpus.reader`

Mètode	Descripció
`fileids()`	Els fitxers del corpus
`fileids([categories])`	Els fitxers del corpus corresponents a aquestes categories
`categories()`	Les categories del corpus
`categories([fileids])`	Les categories del corpus corresponents a aquests fitxers
`raw()`	El contingut en brut del corpus
`raw(fileids=[f1,f2,f3])`	El contingut en brut corresponent als fitxers especificats
`raw(categories=[c1,c2])`	El contingut en brut corresponent a les categories especificades
`words()`	Les paraules del corpus
`words(fileids=[f1,f2,f3])`	Les paraules dels fitxers especificats
`words(categories=[c1,c2])`	Les paraules de les categories especificades
`sents()`	Les oracions del corpus
`sents(fileids=[f1,f2,f3])`	Les oracions dels fitxers especificats
`sents(categories=[c1,c2])`	Les oracions de les categories especificades
`abspath(fileid)`	La ubicació de l'arxiu especificat en el disc
`encoding(fileid)`	La codificació de caràcters de l'arxiu especificat (si és coneguda)
`open(fileid)`	Obre un *stream* per a llegir l'arxiu de corpus especificat
`root()`	La ruta de l'arrel del corpus instal·lat a la nostra màquina
`readme()`	El contingut de l'arxiu `README` del corpus

Si disposem d'un corpus propi format per una col·lecció de fitxers de text també el podem llegir mitjançant NLTK i utilitzar el mètodes descrits per als corpus de l'NLTK, fent servir el `PlaintextCorpusReader`. En els fitxers corresponents a aquest mòdul s'inclou una col·lecció de textos en el directori *corpus*. Copieu aquesta carpeta a alguna ubicació que us vagi bé i executeu el programa següent (`programa-3-11.py`), ajustant la variable `directori_corpus`:

```
 -*- coding: utf-8 -*-

from nltk.corpus import PlaintextCorpusReader
directori_corpus='./corpus'
reader = PlaintextCorpusReader(directori_corpus, '.*\.cat',encoding="utf-8")

print "Arxius del corpus: "
print reader.fileids()
print "pitja ENTER"
a=raw_input()

print "Tot el contingut en un string"
tot=reader.raw()
print tot
print "pitja ENTER"
a=raw_input()

print "Totes les paraules del corpus"
paraules=reader.words()
for paraula in paraules:
    print paraula
print "pitja ENTER"
a=raw_input()

print "Totes les oracions del corpus"
oracions=reader.sents()
for oracio in oracions:
    print oracio
print "pitja ENTER"
a=raw_input()

print "Tots els paràgrafs del corpus"
paragrafs=reader.paras()
for paragraf in paragrafs:
    print paragraf
print "pitja ENTER"
a=raw_input()
```

Demostrem el funcionament del `PlaintextCorpusReader` en el programa anterior. Aquesta classe ens pot anar bé per a tractar corpus de text, ja que disposem dels mètodes necessaris per a segmentar el text en paràgrafs, oracions i paraules.

2. Codificació de caràcters. Unicode

En aquest apartat veurem a fons tots els aspectes relacionats amb els diferents codis de caràcters. En informàtica com a norma general les dades estan representades com a octets. Un octet és una unitat d'informació formada per 8 bits i que pot representar un valor numèric comprès entre 0 i 255 ($2^8 = 256$). El concepte d'octet està molt relacionat amb el concepte de byte.

Es poden establir diferents convencions sobre com un octet o una seqüència d'octets representa una dada en concret. Per exemple, sota certs estàndards, quatre octets consecutius sovint representen una unitat que presenta un nombre real. El cas més senzill, i que es fa servir molt sovint, és el que un octet representa un caràcter segons una taula de correspondència. La interpretació correcta implica que es coneix el codi de caràcters que es fa servir.

Aquest apartat és important, ja que ens podem trobar en situacions en què els caràcters no coincidents amb el llatí bàsic es tractin de manera incorrecta. Aquests defectes es produeixen sempre per no conèixer el codi de caràcters dels nostres arxius o bé no saber-lo tractar correctament. Tots els sistemes operatius actuals i la majoria de programes informàtics són capaços de tractar correctament qualsevol caràcter.

2.1. Algunes definicions importants

En aquest subapartat intentarem definir alguns conceptes importants necessaris per a comprendre els principis bàsics de la codificació de caràcters.

Un **repertori de caràcters** (*character repertoire*) és el conjunt de caràcters diferents per representar.

El **codi de caràcters** (*character code*) és una correspondència, normalment presentada en forma tabular, entre els caràcters d'un repertori de caràcters i un conjunt de nombres enters positius. És a dir, s'assigna un codi numèric únic a cada caràcter del repertori.

La **codificació de caràcters** (*character encoding*) és un mètode (algorisme) per a presentar els caràcters digitalment fent una correspondència entre les seqüències de codis de caràcters i les seqüències d'octets. En el cas més simple, a cada caràcter li correspon un nombre enter entre 0 i 255 i aquest es fa servir com a octet. Naturalment, aquesta possibilitat només funciona per a repertoris de caràcters de com a màxim 256 caràcters (quantitat que no és suficient per a totes les llengües; pensem, per exemple, en el xinès).

2.2. Els codis de caràcters més habituals

En aquest subapartat descriurem els codis de caràcters més emprats. Deixarem per al subapartat 2.3. següent tot el que fa referència a Unicode. Així, aquí exposarem els codis següents:

Hi ha molts codis de caràcters. No cal exposar-los tots, sinó entendre bé el mecanisme de funcionament. Aprendrem a reconèixer altres codis de caràcters i a transformar-los en apartats propers.

- ASCII
- La família ISO-8859
- Codis de caràcters de Windows

2.2.1. ASCII

El codi de caràcters ASCII és un codi de 7 bits (128 posicions). Ara estem parlant d'octets o bytes de 8 bits; per tant, ens sobra un bit (el primer). Aquest primer bit es pot fer servir com a bit de paritat o bé per a disposar de 128 posicions addicionals (128-255).

ASCII és l'abreviatura d'American Standard Code for Information Interchange.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

És important saber llegir una taula de caràcters com l'anterior. Per a saber quin codi li correspon al caràcter *c*, per exemple, cal veure que es troba en la fila 6 i en la columna 3 i per tant li correspon el codi 63 hexadecimal, i el caràcter *M* és el 4D hexadecimal. Si volem saber l'equivalència en decimal, podem escriure en una sessió interactiva de Python:

```
>>> print int("63",16)
99
>>> print int("4D",16)
77
```

Podem verificar les equivalències fent:

```
>>> print chr(99)
c
>>> print chr(77)
M
```

I conèixer directament la codificació d'un determinat caràcter:

```
>>> print ord(“c“)
99
>>> print ord("M")
77
```

2.2.2. La família ISO-8859

La part baixa de les taules de la família ISO-8859 és igual que la de l'ASCII. La part alta es fa servir per a codificar els caràcters no inclosos en el llatí bàsic. A la taula 2 presentem les diferents taules corresponents a l'ISO-8859.

Taula 2. Família ISO-8859

Família	Alfabet	Idiomes
ISO-8859-1	Alfabet llatí núm. 1	Europeu occidental
ISO-8859-2	Alfabet llatí núm. 2	Europeu central i de l'est
ISO-8859-3	Alfabet llatí núm. 3	Europeu del sud (maltès i esperanto)
ISO-8859-4	Alfabet llatí núm. 4	Europeu del nord
ISO-8859-5	Alfabet ciríl·lic	Per a llengües eslaves ciríl·liques
ISO-8859-6	Alfabet àrab	Àrab
ISO-8859-7	Alfabet grec	Per a grec modern
ISO-8859-8	Alfabet hebreu	Per a hebreu i jiddisch
ISO-8859-9	Alfabet llatí núm. 5	Turc
ISO-8859-10	Alfabet llatí núm. 6	Nòrdic (finès, esquimal, islandès)
ISO-8859-11	Alfabet thai	Llengua thai
ISO-8859-12	No està definit	
ISO-8859-13	Alfabet llatí núm. 7	Llengües bàltiques
ISO-8859-14	Alfabet llatí núm. 8	Celta
ISO-8859-15	Alfabet llatí núm. 9	Actualització del llatí 1 amb el símbol de l'euro i altres
ISO-8859-16	Alfabet llatí núm. 10	Per a diverses llengües

A la figura 1 podeu veure la taula de caràcters corresponent a l'ISO-8859-1 i a la figura 2 la corresponent a l'ISO-8859-5. En aquestes taules només es representa la part alta, ja que la part baixa coincideix amb l'ASCII.

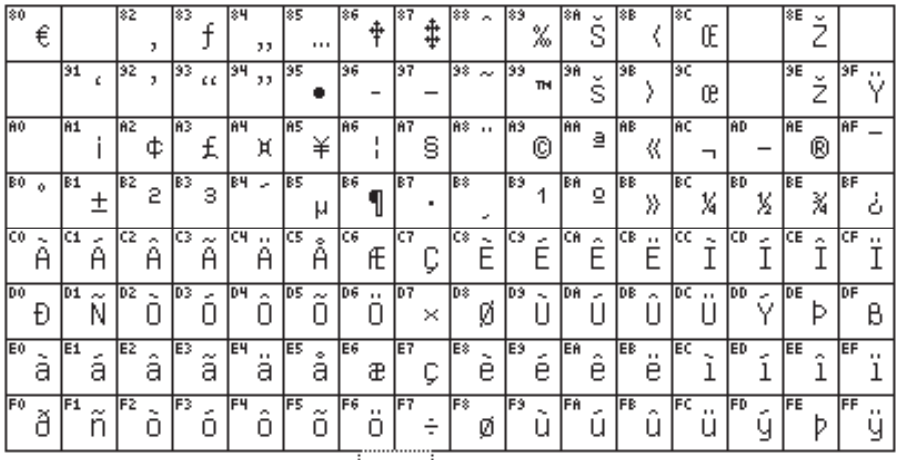
Figura 1. Taula de caràcters de la família ISO-8859-1

Figura 2. Taula de caràcters de la família ISO-8859-5

2.2.3. Codis de caràcters de Windows

Microsoft va desenvolupar una codificació pròpia, diferent de l'ISO-8859. A la figura 3 podeu observar la codificació WinLatin1 o *Windows code page 1252*. Si ens fixeu en les diferències amb l'ISO-8859-1, són que aprofita algunes posicions que en l'ISO no es fan servir i d'aquesta manera pot codificar més caràcters.

Figura 3. WinLatin 1 o *Windows code page 1252*



Hi ha codis de caràcters per a altres alfabets, concretament:

- 1250 (Europa central)
- 1251 (Ciríl·lic)
- 1252 (Llatí I)
- 1253 (Grec)
- 1254 (Turc)
- 1255 (Hebreu)
- 1256 (Àrab)
- 1257 (Bàltic)
- 1258 (Vietnamita)
- 874 (Thai)

2.3. Unicode

En el subapartat 2.2. hem presentat una sèrie de codis de caràcters que fan servir 8 bits. Això dóna la possibilitat de codificar fins a 256 caràcters. Per a molts idiomes això és suficient, però no per tots (penseu, per exemple en els caràcters xinesos). Tot i que pugui ser suficient per a molts idiomes, fa que sigui impossible guardar en un únic arxiu de text* documents multilingües (per exemple, barrejar en un únic document català i rus). També s'ha de tenir en compte que de tant en tant apareixen nous símbols (pensem per exemple en el símbol de l'euro) que s'han d'anar incorporant al codi de caràcters.

***Fixeu-vos que diem "de text". Fent servir altres formats sí que és possible.**

En el subapartat 2.2. hem vist uns quants codis de caràcters dels molts existents. Aquesta gran quantitat de codis de caràcters implica la dificultat d'obrir un document correctament, ja que la detecció del codi de caràcters no és totalment automàtica. Per aquests motius s'intenta adoptar un codi de caràcters universal. Aquest codi de caràcters és **Unicode**. Unicode fa servir més de 8 bits, de manera que pot codificar molts més caràcters.

Originàriament es pensava fer servir simplement una codificació de 16 bits, que proporciona la possibilitat de codificar més de 65.000 caràcters (2^{16} = 65.536). Tot i que aquesta xifra és suficient per a codificar la majoria dels milers de caràcters que es fan servir a les diferents llengües del món, l'estàndard Unicode ISO/IEC 10646 permet tres formes de codificació que fan servir un repertori de caràcters comú però que permeten codificar al voltant d'un milió més de caràcters. Aquesta xifra és suficient per a cobrir totes les necessitats de codificació conegudes, incloent-hi totes les escriptures històriques del món i altres sistemes de notació.

Adreça web recomanada

A l'adreça *http://www.unicode.org/charts* podeu consultar totes les taules de caràcters Unicode.

Per la seva extensió seria impossible reproduir totes les taules de caràcters Unicode aquí. A la figura 4 podeu veure una part de la taula corresponent a les posicions reservades als ideogrames comuns del xinès, japonès i coreà.

Figura 4. Part de la taula Unicode dels ideogrames CJK unificats

2.3.1. Codificacions de caràcters amb Unicode

Hi ha diferents maneres de codificar els caràcters amb Unicode. La majoria d'ordinadors fan servir unitats mínimes de 8 bits. Si fem servir més de 8 bits haurem d'organitzar la codificació de manera que fem servir múltiples de 8 bits, és a dir, més d'un byte. L'estàndard Unicode defineix tres tipus de codificacions que permeten representar la informació en un byte, dos, tres o quatre bytes. Les tres codificacions codifiquen el mateix repertori de caràcters comú i es pot passar d'una codificació a una altra sense pèrdua de dades.

- **UTF-8**: la codificació en bytes és d'una longitud variable, des d'1 byte per als caràcters coincidents amb l'ASCII.
- **UTF-16**: la codificació també és variable, però o bé en dos bytes o bé en quatre.
- **UTF-32**: tots els caràcters es codifiquen amb quatre bytes.

2.4. Tractament de les codificacions de caràcters amb Python

Python pot treballar perfectament amb qualsevol codificació de caràcters, tot i que Python disposa d'una codificació específica que s'anomena *unicode-escape* i internament sempre treballa amb aquesta codificació. Aquesta codificació converteix tots els caràcters que no siguin ASCII en la seva representació *\xXX*, si el seu codi està entre 128 i 256, i amb *\uXXXX* per a posicions superiors.

En l'exemple següent (`codificacio-1.py`) obrim un fitxer de text que està en Unicode UTF-8 i l'escrivim per pantalla, primer amb la codificació interna de Python (*unicode-escape*), després en UTF-8 i finalment en ISO-8859-1. Quina sortida per pantalla veieu correctament en el vostre ordinador?

```
import codecs
arxiu = codecs.open(”text1.txt”, encoding=”utf-8”)
print ”\nunicode-escape”
for linia in arxiu:
    print linia.encode(’unicode-escape’)

arxiu = codecs.open(”text1.txt”, encoding=”utf-8”)
print ”\nunicode-utf-8”
for linia in arxiu:
    print linia.encode(’utf-8’)

arxiu = codecs.open(”text1.txt”, encoding=”utf-8”)
print ”\niso-8859-1”
for linia in arxiu:
    print linia.encode(’iso-8859-1’)

arxiu = codecs.open(”text1.txt”, encoding=”utf-8”)
print ”\nlatin-1”
for linia in arxiu:
    print linia.encode(’latin-1’)
```

El mòdul `codecs` proporciona una sèrie de funcions per a llegir dades codificades en cadenes Unicode i per a escriure cadenes Unicode en diferents codificacions de caràcters. La funció `codecs.open()` té com a paràmetres el fitxer per obrir i la codificació d'aquest fitxer. En l'exemple anterior el fitxer està codificat en Unicode UTF-8. Quan fem `print linia` especifiquem la codificació de caràcters que desitgem amb `encode()`.

En l'exemple següent (`codificacio-2.py`) fem el mateix amb un text en polonès que prové d'una col·lecció de textos d'exemple de l'NLTK i les escrivim en *unicode-escape* i UTF-8. Podeu veure correctament els caràcters especials?

```
import nltk
import codecs
ruta = nltk.data.find(’corpora/unicode_samples/polish-lat2.txt’)
arxiu = codecs.open(str(ruta), encoding=’latin2’)
print "\nunicode-escape"
for linia in arxiu:
    print linia.encode(’unicode_escape’)

arxiu = codecs.open(str(ruta), encoding=’latin2’)
print "\nunicode-utf-8"
for linia in arxiu:
    print linia.encode(’utf-8’)
```

El mòdul `unicode data` ens permet obtenir informació sobre els caràcters Unicode. En el programa següent (`codificacio-3.py`) podem veure tots els caràcters Unicode des de la posició 0 a la posició 100.000 i la seva descripció. Podeu veure correctament els diferents caràcters per pantalla?

```
import unicodedata
for i in range(0,100000):
    caracter=unichr(i)
    try:
        descripcio=unicodedata.name(caracter)
    except(Exception):
        descripcio="NO DESCRIPTION"
    finally:
        print str(i)+" "+caracter+" "+descripcio
```

Modifiqueu el `range(0,10000)` per un rang de 0370 a 03FF en hexadecimal (`codificacio-3b.py`). Fixeu-vos que podem expressar nombres hexadecimals amb `int('0370',16)`.

```
import unicodedata
for i in range(int('0370',16),int('03FF',16)):
    caracter=unichr(i)
    try:
        descripcio=unicodedata.name(caracter)
    except(Exception):
        descripcio="NO DESCRIPTION"
    finally:
        print str(i)+" "+caracter+" "+descripcio
```

Ara hauríeu de poder visualitzar la taula de caràcters que mostrem a la figura 5. Si no és així, escriviu la sortida del programa en un arxiu que s'anomeni `sortida.txt`. Si obriu el fitxer `sortida.txt` amb un editor adequat podreu observar els caràcters correctament.

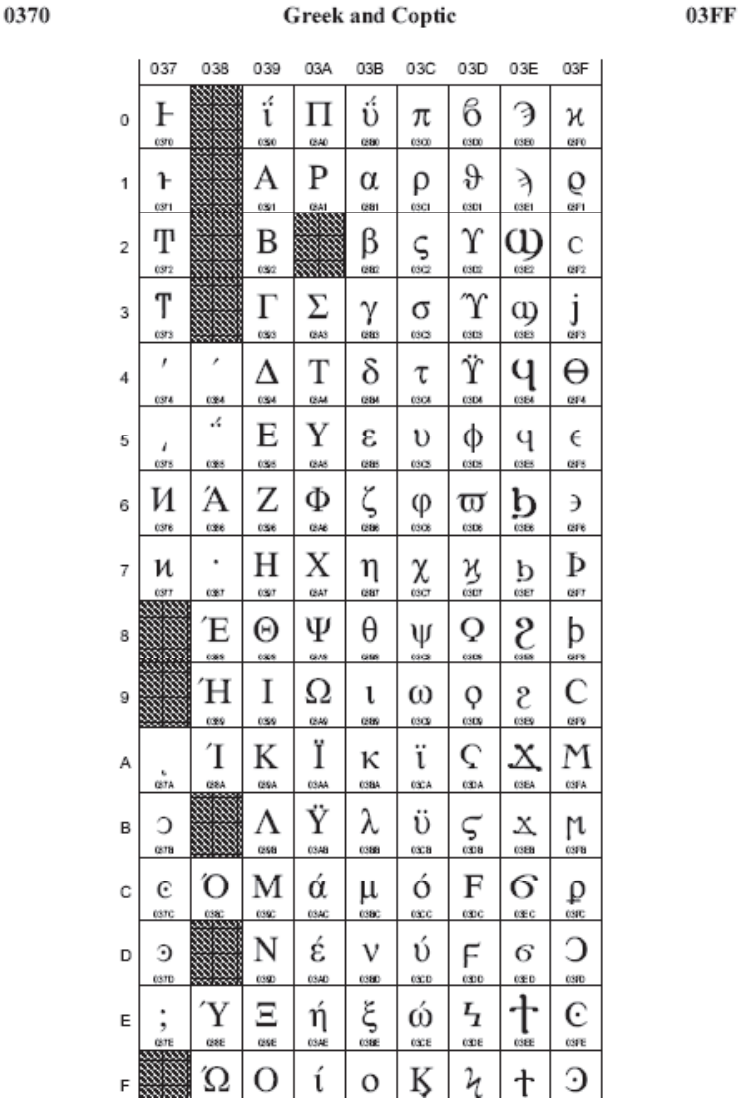
Fins aquí hem vist com podem obrir correctament amb Python un fitxer en una codificació correcta, però suposant que coneixem prèviament la codificació de caràcters en què està escrit. Python també pot intentar esbrinar la codificació de caràcters d'un determinat document.

```
>>> from nltk.util import guess_encoding
>>> fitxer=open("noticia1.txt","rU")
>>> linies=fitxer.readlines()
>>> (decodificat,codificacio)=guess_encoding(text)
>>> print codificacio
```

```
utf-8
>>> print decodificat
La ferida de l'11-M s'amplia

Els reis presideixen la inauguració del monument a les víctimes
...
```

Figura 5. Caràcters grecs i còptics d'Unicode



3. Segmentació en unitats lèxiques: *tokenització*

La segmentació en unitats lèxiques o *tokenització*, consisteix a dividir el text en unitats més petites (que sovint coincideixen amb paraules). Tot i que aquesta és una tasca molt bàsica i necessària per a poder portar a terme tasques d'anàlisi més avançades, aquesta tasca presenta nombrosos problemes que no són fàcils de solucionar. Hi ha nombrosos treballs sobre aquesta àrea entre els quals es poden destacar els treballs de Grefenstette i Tapanainen (1994) i Mikheev (2002). Sigui com sigui, actualment es pot considerar que aquesta tasca es resol de manera satisfactòria. En aquest apartat aprendrem algunes tècniques per a segmentar el text en unitats lèxiques i anirem observant els diferents problemes que hi apareixen i com es poden solucionar.

Quines unitats o *tokens* volem detectar dependrà una mica de la tasca que voldrem fer després. En general, pel que fa als exemples següents, ens interessarà obtenir les paraules separades i també els signes de puntuació. Per exemple, d'una frase com:

Hola bon dia, avui fa sol.

Voldrem obtenir els *tokens* ["Hola", "bon", "dia", ",", "avui", "fa", "sol", "."]. Però d'una frase com:

El Sr. Joan vol dir-nos la seva opinió.

No voldrem separar "Sr" i ".", sinó mantenir-los junts "Sr.".

La tokenització és una tasca bàsica del processament del llenguatge natural, però que presenta nombroses dificultats. Per exemple, el tractament de les xifres, dates, etc. Cal tenir present què és el que volem obtenir i per a quina tasca posterior volem aquests *tokens*.

La primera idea que ens ve al cap és fer servir el mètode `split()` dels `strings` per a obtenir les diferents unitats. En el programa següent podem observar una primera versió molt bàsica del nostre tokenitzador (`programa-3-12.py`).

```
# -*- coding: utf-8 -*-

frase1=”Hola bon dia, avui fa sol”
tokens1=frase1.split()
print tokens1
frase2=”El sr. Joan vol dir-nos la seva opinió.”
tokens2=frase2.split()
print tokens2
```

Fixeu-vos en la primera línia del programa. Estem declarant la codificació de caràcters del fitxer del programa mateix. Com que el nostre programa conté caràcters que no són llatí bàsic és important guardar-lo amb una codificació determinada i coneguda per nosaltres, i declarar aquesta codificació en la primera línia del programa. Una bona opció per a unificar criteris és fer servir Unicode UTF-8.

En les línies següents podem observar la sortida del programa:

```
['Hola', 'bon', 'dia,', 'avui', 'fa', 'sol']
['El', 'sr.', 'Joan', 'vol', 'dir-nos', 'la', 'seva', 'opini\xc3\xb3.']
```

Si ens fixem bé en els resultats del programa anterior, observarem que la divisió en unitats lèxiques (tokenització) no s'ha fet correctament. Podeu comentar alguns casos al Fòrum?

Ara veurem un exemple de programa de tokenització que funciona amb expressions regulars (`programa-3-13.py`).

```
# -*- coding: utf-8 -*-

import nltk
import re
f=open(”noticia1.txt”,”rU”)
text=f.read()
expressioregular=r'''(?x)
\w+
'''
tokens=nltk.regexp_tokenize(text,expressioregular)
print tokens
```

Si proveu aquest programa, veureu que no funciona bé per al català; un dels motius és que l'expressió regular `\w+`, un o més caràcter alfanumèrics seguits, no preveu els caràcters accentuats del català, ni la *ç*, etc., és a dir, considera com a caràcters alfabètics únicament els llatins bàsics. A banda d'això segura-

ment, en comptes de veure els caràcters accentuats veureu el codi associat. Per a arreglar el primer punt podeu canviar aquesta línia per:

```
[\wàèòéíóúïüçÀÈÒÉÍÓÚÏÜÇ]+
```

El programa modificat el trobareu a `programa-3-13b.py`.

En comptes de modificar l'expressió regular, podem indicar a Python que el nostre arxiu està en una codificació específica i d'aquesta manera podem fer servir `\w+` sense problemes, això ho podem veure en el `programa-3-13b2.py`.

```
# -*- coding: utf-8 -*-

import nltk
import re
f=open("noticia1.txt","rU")
text=f.read()
text=text.decode('utf-8')
expressioregular=r'''(?x)
[\w]
'''
tokens=nltk.regexp_tokenize(text,expressioregular)
print tokens
```

Però la tokenització encara no funcionarà correctament. Per exemple, un altre problema és que detecta les paraules, però elimina els signes de puntuació. Si ens interessa tenir també els signes de puntuació com a *tokens* hem d'introduir la modificació següent (programa-3-13c.py):

```
expressioregular=r'''(?x)
\w+|
[\,\;\.\:\!\?\(\)]
'''
```

Potser aquesta solució tampoc no és la millor, però la nostra tokenització va millorant.

Durant uns dies, anirem proposant al Fòrum de l'aula millores per a arribar a trobar una sèrie d'expressions regulars que funcionin bé per al català.

Podeu provar amb altres arxius modificant la línia on es carrega l'arxiu* o fins i tot donant un valor concret a la variable `text`, sense carregar-la de cap arxiu (trobareu el programa a `programa-3-13d.py`):

***L'arxiu ha d'estar al mateix directori o bé cal indicar la ruta sencera a l'arxiu.**

```
text='Avui fa un dia molt bonic i el sr. Arnau sortirà a passejar a les 13:45.'
```

3.1. Escriure la sortida a un fitxer

Si observeu els resultats per la pantalla de l'ordinador, depenent de la configuració del vostre ordinador és possible que no vegeu bé els caràcters accentuats i altres caràcters no coincidents amb el llatí bàsic. Una bona solució és escriure els resultats a un fitxer. Un cop executat el programa obriu el fitxer de sortida amb un editor de textos. Podeu trobar un exemple en el programa següent (`programa-3-14.py`).

```
# -*- coding: utf-8 -*-

import nltk
import re
import os, sys
import string
f=open("noticia1.txt","rU")
w=open("sortida.txt","w")
text=f.read()
text=text.decode('utf-8')
expressioregular=r'''(?x)
\w+|
[\,\;\.\:\!\?\(\)]
'''
tokens=nltk.regexp_tokenize(text,expressioregular)
w.write(string.join(list(tokens),"\n").encode('utf-8'))
```

Amb la instrucció `f=open("sortida.txt","w")` obrim un fitxer en mode d'escriptura que es diu `sortida.txt`.

`w.write(string.join(list(tokens),"\n").encode('utf-8'))` és una instrucció que escriu en aquest fitxer la llista de paraules tokenitzades, però unides mitjançant un salt de línia (*\n*). D'aquesta manera el resultat és més visual.

Obriu el fitxer de sortida. El text ha estat correctament tokenitzat? Un dels problemes és que els apòstrofs no es veuen i les paraules apostrofades apareixen sense l'apòstrof. Modifiqueu l'expressió regular (`programa-3-14b.py`) de la manera següent:

```
\w+'?
```

La tokenització és correcta, ara? Feu les vostres aportacions al Fòrum.

3.2. El lector de corpus de text pla de l'NLTK

NLTK proporciona una classe anomenada `PlaintextCorpusReader` que funciona com un lector de documents en text pla. Aquesta classe ens permet accedir als diferents paràgrafs del corpus (que se suposa que estan separats per línies en blanc) i proporciona mètodes per a tokenitzar i segmentar el corpus. La tokenització i segmentació es pot dur a terme a partir dels que proporciona la classe per defecte o be personalitzar-los segons les nostres necessitats.

Aquesta classe ens permetrà llegir corpus i fer tasques bàsiques d'una manera senzilla. En el programa següent llegim un corpus i mostrem el text, els *tokens*, els segments i els paràgrafs (`lector.py`):

```
from nltk.corpus import PlaintextCorpusReader
rutacorpus="."
lector=PlaintextCorpusReader(rutacorpus,'corpus-DOGC-cat.txt',encoding='utf8')

print "TOT EL TEXT"
print "pitja ENTER"
a=raw_input()

toteltext=lector.raw()
print toteltext
print "pitja ENTER"
a=raw_input()

print "TOKENITZAT"
print "pitja ENTER"
a=raw_input()

tokens=lector.words()
for token in tokens:
    print token
print "pitja ENTER"
a=raw_input()

print "SEGMENTAT"
print "pitja ENTER"
a=raw_input()

frases=lector.sents()
for frase in frases:
    print frase
print "pitja ENTER"
a=raw_input()
```

```
print "PARAGRAFS"
print "pitja ENTER"
a=raw_input()

paragrafs=lector.paras()
for paragraf in paragrafs:
    print paragraf
print "pitja ENTER"
a=raw_input()
```

Si ens fixem bé en els resultats de la tokenització podem veure alguns errors, com per exemple:

```
Superfície
d
'
ocupació
temporal
,
en
m2
:
1
.
497
.
```

En la segmentació també podem trobar diversos errors:

```
...u'processal', u'del', u'SR', u'.']
[u'MANUEL', u'PEREZ', u'ARNAU', u'i', u'haig', u'de', u'condemnar', u'i',
u'condemno', u'el', u'SR', u'.']
[u'AGUSTIN', u'JAUSAS', u'MARTI', u'i', u'els', u'IGNORATS', u'HEREUS', u'DE',
u'JUAN', u'JAUSAS', u'MARTI', u'a', u'elevar', u'a', u'escriptura',
u'p\xfablica', u'el', u'document', u'privat', u'de', u'venda',
u'de', u'data', u'1', u'de', u'juliol',...
```

Com hem comentat, és possible personalitzar el tokenitzador i el segmentador per utilitzar. En el següent (`lector2.py`) hem personalitzat el tokenitzador mitjançant un tokenitzador que funciona amb expressions regulars:

```
from nltk.corpus import PlaintextCorpusReader
from nltk.corpus import RegexpTokenizer
rutacorpus="."
```

```
expregtoken=r'''(?x)
\d+\.\d+|
\w+\'?|
[\,\;\.\:\!\?\(\)]
'''
tokenitzador=RegexpTokenizer(expregtoken)

lector=PlaintextCorpusReader(rutacorpus,'corpus-DOGC-cat.txt',
       encoding='utf8',word_tokenizer=tokenitzador)


tokens=lector.words()
for token in tokens:
    print token
```

Com podem observar, el funcionament ha millorat. Observem com actua en el mateix segment que presentàvem com a error:

```
Superfície
d'
ocupació
temporal
,
en
m2
:
1.497
.
```

Segur, però, que encara queden coses per millorar i afinar.

Exercici

Intenteu personalitzar el segmentador en oracions. Per a fer-ho, heu de crear un segmentador i passar-lo com a paràmetre `sent_tokenizer`.

4. Segmentació del text

En aquest apartat tractarem un problema semblant al de la tokenització: la segmentació del text en oracions. Aquesta també és una tasca bàsica del processament del llenguatge natural que ha estat àmpliament estudiada. Podeu trobar una descripció detallada de les tasques de segmentació i tokenització a Palmer (2000). La idea bàsica és semblant a la de la tokenització, però ara l'expressió regular que farem servir canviarà. Una primera opció seria fer servir una expressió regular com la següent:

```
expressioregular=re.compile(r'''
.+?[\.\?\!][\s]
| .+?\n
''',re.VERBOSE)
```

Proveu aquesta expressió regular (`programa-3-15.py`) i proposeu millores. El resultat de la segmentació el podeu trobar en l'arxiu `sortida.txt`. Què passa si intenteu segmentar un text com el següent? (aquest text el trobareu en el fitxer `text1.txt`; modifiqueu el programa perquè obri aquest fitxer):

```
J.A. Capetti arribarà amb la R.E.N.F.E. A les 16:20. Marxarà el 22 de novembre
amb el vol LU-345. Contacta amb ell a jacapetti@miami.mafia.com, i digue'm
alguna cosa. Les apostes estan 4:1 i cal evitar que pugin més. Hi ha cent milions
de pts. en joc. El Sr. Capetti és el "capo" a Miami i hi hem de quedar bé.
```

Una altra opció per a segmentar el text consisteix a introduir en el text unes marques de salt (per exemple `@BREAK@`) i de no salt (per exemple `@NOBREAK@`). Després segmentarem el text per les marques de salt i finalment esborrarem totes aquestes marques del text. Aquesta aproximació ens proporciona una mica més de control sobre tot el procés, ja que podem afegir diferents expressions regulars d'una manera una mica més controlada. En l'exemple següent (`programa-3-16.py`) podem veure aquesta aproximació.

```
# -*- coding: utf-8 -*-

import nltk
import re
import os, sys
import string
```

```
f=open(”text1.txt”,”rU”)
w=open(”sortida.txt”,”w”)
text=f.read()

#expressions de NO salt d’oració (@NOBREAK@)

#abreviatures (caldria completar la llista)
text=re.sub(r’Sr\.’, r’Sr.@NOBREAK@’,text)
text=re.sub(r’pts\.’, r’pts.@NOBREAK@’,text)

#sigles
text=re.sub(r’(([A-Z]+\.)+)’, r’\1@NOBREAK@’,text)

#expression de salt d’oració (@BREAK@)

#salt per punt, interrogant i exclamació seguit d’espai
text=re.sub(r’([\.|\?|\!])[\s]’,r’.@BREAK@’, text)

#salt per final de linia
rebreakfinal=re.compile(r’’’$’’’,re.VERBOSE)
text=rebreakfinal.sub(’@BREAK@’, text)

#Escrivim el text per pantalla simplement per veure com ha posat les marques
print ”TEXT AMB MARQUES”
print text

#segmentem per salt
expressioregular=r’’’(?x)
.+?@BREAK@
’’’

#segmentem
segments=nltk.regexp_tokenize(text,expressioregular)

#escrivim al fitxer el text segmentat

for segment in segments:
segment=re.sub(r’@NOBREAK@’,r’’,segment)
segment=re.sub(r’@BREAK@’,r’ ’,segment)
w.write(segment+”\n”)
```

Us animo també a fer les vostres aportacions de millora d’aquest segmentador al Fòrum.

5. Freqüències i distribucions de freqüència

Fins ara en aquest mòdul hem fet algunes tasques de processament sobre corpus i textos, però encara no hem fet cap tipus de càlcul sobre les dades processades. En aquest apartat aprendrem a fer alguns càlculs senzills sobre corpus: freqüències absolutes i freqüències relatives, distribucions de freqüència i a trobar les col·locacions més freqüents d'un corpus.

5.1. Freqüència absoluta

Entenem per **freqüència absoluta** el nombre total de vegades que apareix una determinada unitat lèxica en el nostre corpus.

El càlcul de la freqüència absoluta d'una paraula és senzill: podem utilitzar un diccionari per a posar com a clau les paraules i anar incrementant el valor del diccionari cada cop que apareix la paraula. En el programa següent (`programa-3-17.py`) podem veure una implementació senzilla d'aquesta idea:

```
from nltk.corpus import cess_cat

paraules=cess_cat.words()

freq={}

for paraula in paraules:
    if (freq.has_key(paraula)):
        freq[paraula]+=1
    else:
        freq[paraula]=1

for paraula in freq.keys():
    print str(freq[paraula])+"\t"+paraula
```

Com que el càlcul de freqüències és una operació molt habitual en el processament del llenguatge natural, NLTK proporciona la classe `FreqDist` que calcula les vegades que apareix cada paraula d'una llista. A continuació podeu

veure el mateix programa anterior implementat fent ús de la següent funció (`programa-3-17b.py`):

```
from nltk.corpus import cess_cat
from nltk import FreqDist

paraules=cess_cat.words()

fd=FreqDist(paraules)

for paraula in fd.keys():
    print str(fd[paraula])+"\t"+paraula
```

Com que les sortides de tots dos programes presentats anteriorment són molt llargues, pot ser una bona idea treure els resultats en un fitxer i obrir-lo per a poder veure la sortida en detall. Quina diferència bàsica trobeu entre les dues sortides? Com veureu, en el segon cas la sortida està endreçada de més freqüència a menys freqüència.

`FreqDist` de NLTK ens proporciona una sèrie de mètodes molt útils, que podeu veure detallats a la taula 3.

Taula 3. Mètodes de la classe `FreqDist` de NLTK

Exemple	Descripció
`fdist = FreqDist(mostres)`	Crea una distribució de freqüència que conté les mostres donades
`fdist.inc('paraula')`	Incrementa el comptatge per `'paraula'`
`fdist['paraula']`	Compta el nombre de vegades que apareix `'paraula'`
`fdist.freq('paraula')`	Dóna la freqüència relativa de `'paraula'`
`fdist.N()`	Dóna el nombre total de paraules
`fdist.keys()`	Dóna les paraules en ordre de freqüència decreixent
`for paraula in fdist:`	Itera sobre les paraules en ordre decreixent de freqüència
`fdist.max()`	Dóna la paraula amb freqüència màxima
`fdist.tabulate()`	Tabula la distribució de freqüència
`fdist.plot()`	Grafica la distribució de freqüència
`fdist.plot(cumulative=True)`	Grafica la distribució de freqüència de manera acumulativa
`fdist1 < fdist2`	Prova si les mostres de `fdist1` apareixen amb menys freqüència que en `fdist2`

5.2. Freqüència relativa

La freqüència absoluta d'una paraula en un determinat corpus no ens dóna informació real sobre si la paraula és molt freqüent o no, perquè això dependrà de la mida del corpus. Que una paraula aparegui, diguem-ne, 22 vegades en en nostre corpus, no ens diu res, ja que si el corpus és molt gran potser aquest valor de freqüència és petit.

La **freqüència relativa** d'una paraula en un corpus és el nombre de vegades que hi apareix dividida pel nombre total de paraules en el corpus.

En una sessió interactiva de Python feu el següent*:

*S'han numerat les línies per a facilitar l'explicació posterior.

```
(1)>>> import nltk
(2)>>> from nltk.corpus import cess_cat
(3)>>> paraules=cess_cat.words()
(4)>>> fdist=nltk.FreqDist(paraules)
(5)>>> fdist['haver']
315
(6)>>> fdist.freq('haver')
0.00062518234485058142
(7)>>> fdist.N()
503853
(8)>>> 315/503653
0
(9)>>> 315.0/503653
0.000625430604007124
```

En (3) guardem en `paraules` les paraules del corpus i en (4) calculem la distribució de freqüència. En (5) calculem la freqüència absoluta d'*haver* i en (6) la relativa. En (7) obtenim les paraules totals del corpus i en (9) intentem calcular la freqüència relativa dividint la freqüència absoluta pel nombre de paraules totals. Veiem que dóna zero: el motiu és la precisió del càlcul, ja que el resultat és massa baix per a la precisió que tenim en dividir un enter entre un altre nombre tan gran. En (9) hem forçat que la freqüència absoluta tingui una precisió més gran i el resultat que obtenim és pràcticament igual que el que obteníem amb el càlcul de la freqüència relativa directa.

La freqüència relativa de paraules que són força freqüents (com per exemple algunes paraules funcionals) es mantindrà pràcticament constant per a subconjunts del corpus que siguin prou grans. En les següents línies (continuant amb la mateixa sessió interactiva anterior), podem observar aquest fenomen:

```
>>> paraules1=paraules[:100000]
>>> paraules2=paraules[100000:200000]
>>> paraules3=paraules[200000:300000]
>>> fdist1=nltk.FreqDist(paraules1)
>>> fdist2=nltk.FreqDist(paraules2)
>>> fdist3=nltk.FreqDist(paraules3)
>>> fdist.freq("de")
0.047005773509337045
>>> fdist1.freq("de")
```

```
0.05289
>>> fdist2.freq(”de”)
0.04172
>>> fdist3.freq(”de”)
0.046089999999999999
```

En canvi, per a paraules poc freqüents, aquest fenomen no es complirà:

```
>>> fdist.freq(”avaluat”)
1.9847058566685126e-06
>>> fdist1.freq(”avaluat”)
1.0000000000000001e-05
>>> fdist2.freq(”avaluat”)
0.0
>>> fdist3.freq(”avaluat”)
0.0
```

5.3. La llei de Zipf

La Llei de Zipf afirma que donat un corpus, la freqüència d'una paraula és inversament proporcional a la seva posició a la taula de freqüències (*rank*).

Amb la seva llei, Zipf (1949) afirma que hi ha una constant k que es pot calcular multiplicant la freqüència de qualsevol paraula per la seva posició a la taula (*rank*) ($k = f \cdot r$).

En el programa següent (`zipf.py`) avaluem la llei de Zipf amb les 50 paraules més freqüents del corpus `Cess_cat`:

```
import nltk
from nltk.corpus import cess_cat
import re

paraules=cess_cat.words()
freqdist=nltk.FreqDist(paraules)

posicio=0
p = re.compile('\w+')
for paraula in freqdist.keys():
    if p.match(paraula):
        posicio+=1
```

```
    freq=freqdist[paraula]
    fxr=posicio*freq
    print paraula+"\t"+str(freq)+"\t"+str(posicio)+"\t"+str(fxr)
    if (posicio==50): break
```

Els resultats de l'execució d'aquest programa els podem observar a la taula 4.

Tot i que la llei només mostra una tendència i no és exacta, recalca el fet que en un corpus hi ha molt poques paraules molt freqüents i moltes paraules poc freqüents. En el programa següent (`zipf2.py`) grafiquem aquest fenomen amb un subconjunt del corpus `Cess_cat` de 1.000 paraules. Com podem observar al gràfic (figura 6) el principi esmentat, és a dir, que molt poques paraules apareixen moltes vegades i que moltes apareixen poques vegades, s'acompleix.

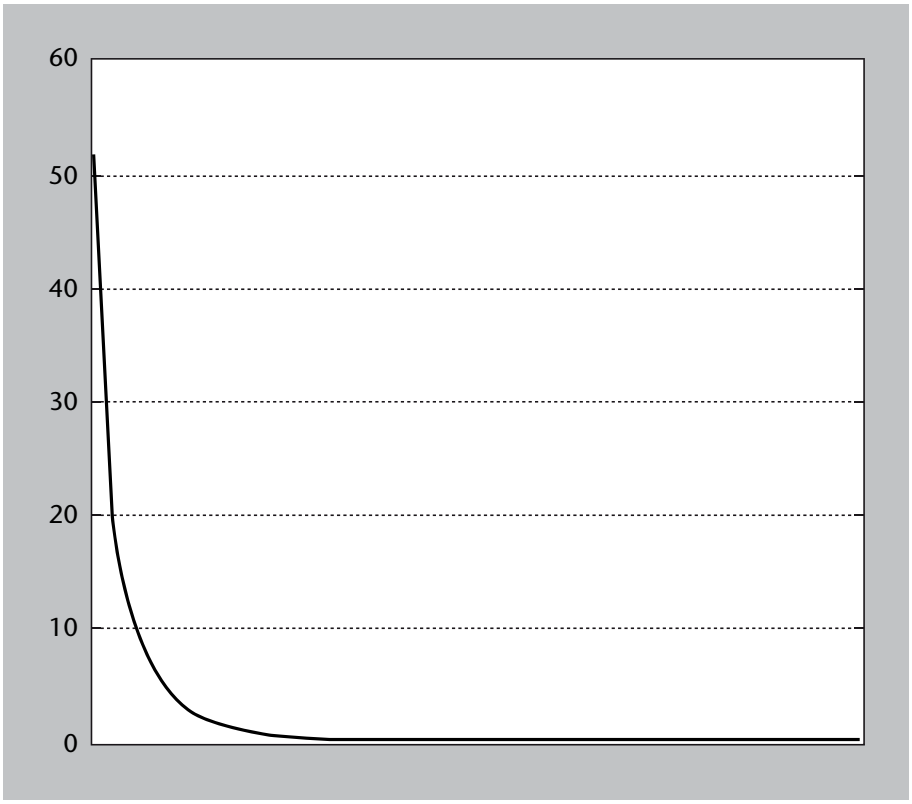
Matplotlib

Per a poder executar aquest programa és imprescindible tenir instal·lat el paquet Matplotlib. A *http://www.nltk.org/download* s'explica com el podem obtenir i instal·lar.

```
import nltk
from nltk.corpus import cess_cat

paraules=cess_cat.words()
paraules1=paraules[:1000]
freqdist=nltk.FreqDist(paraules1)
freqdist.plot()
```

Figura 6. Gràfic de freqüències



Taula 4. Demostració de la llei de Zipf amb les 50 paraules més freqüents del corpus `Cess_cat`

Paraula	Freqüència	*rank*	$f \cdot r$
de	23684	1	23684
la	15695	2	31390
que	12703	3	38109
i	11819	4	47276
el	9749	5	48745
a	9568	6	57408
l'	8892	7	62244
d'	6970	8	55760
del	5826	9	52434
en	5783	10	57830
per	5378	11	59158
un	5247	12	62964
les	5219	13	67847
va	5217	14	73038
ha	5114	15	76710
els	4207	16	67312
una	3895	17	66215
amb	3755	18	67590
es	3265	19	62035
al	2843	20	56860
no	2646	21	55566
El	2551	22	56122
dels	2445	23	56235
s'	2327	24	55848
La	1747	25	43675
més	1691	26	43966
van	1589	27	42903
han	1363	28	38164
aquest	1354	29	39266
hi	1202	30	36060
la_seva	1177	31	36487
L'	1124	32	35968
com	1055	33	34815
aquesta	1047	34	35598
ser	1009	35	35315
als	930	36	33480
també	864	37	31968
entre	862	38	32756
anys	833	39	32487
o	787	40	31480
Els	780	41	31980
fer	690	42	28980
però	685	43	29455
estat	678	44	29832
sobre	669	45	30105
En	662	46	30452
pel	642	47	30174
milions	638	48	30624
perquè	625	49	30625
com_a	607	50	30350

6. Càlcul d'*n*-grames

Donada una determinada oració *Avui fa un dia molt bonic però diuen que demà plourà*, podem calcular les combinacions de dues paraules seguides: *Avui fa, fa un, un dia, dia molt, molt bonic, bonic però, però diuen, diuen que, que demà, demà plourà*.

També podem calcular combinacions de tres paraules seguides: *Avui fa un, fa un dia, un dia molt, dia molt bonic, molt bonic però, bonic però diuen, però diuen que, diuen que plourà*.

Aquestes combinacions s'anomenen normalment *n*-grames, en què la *n* indica el nombre de paraules que combinem. Així, en el primer cas estem parlant de **bigrames** i en el segons cas de **trigrames**.

NLTK ens permet calcular fàcilment els *n*-grames d'una seqüència (per exemple d'una llista). En el programa següent (`ngrams1.py`) en podem veure un exemple:

```
# -*- coding: utf-8 -*-

from nltk.util import ngrams

frase="Avui fa un dia molt bonic però diuen que demà plourà"
tokens=frase.split()
bigrams=ngrams(tokens,2)
print "BIGRAMS"
print bigrams
trigrams=ngrams(tokens,3)
print "TRIGRAMS"
print trigrams
```

La sortida d'aquest programa seria la següent:

```
BIGRAMS
[('Avui', 'fa'), ('fa', 'un'), ('un', 'dia'), ('dia', 'molt'),
('molt', 'bonic'), ('bonic', 'per\xc3\xb2'), ('per\xc3\xb2', 'diuen'),
('diuen', 'que'), ('que', 'dem\xc3\xa0'), ('dem\xc3\xa0', 'plour\xc3\xa0')]
TRIGRAMS
[('Avui', 'fa', 'un'), ('fa', 'un', 'dia'), ('un', 'dia', 'molt'),
```

```
('dia', 'molt', 'bonic'), ('molt', 'bonic', 'per\xc3\xb2'),
('bonic', 'per\xc3\xb2', 'diuen'), ('per\xc3\xb2', 'diuen', 'que'),
('diuen', 'que', 'dem\xc3\xa0'), ('que', 'dem\xc3\xa0', 'plour\xc3\xa0')]
```

Els *n*-grames tenen diverses aplicacions dins del processament del llenguatge natural. A continuació en mostrem tres: el càlcul de col·locacions, l'extracció automàtica de terminologia i la creació de models del llenguatge.

6.1. Càlcul de col·locacions

Una col·locació són dues o més paraules que apareixen juntes sovint, és a dir, que apareixen juntes amb més freqüència de la que es podria esperar de la casualitat. En el programa següent, `colocacions1.py`, calculem les 25 col·locacions més probables d'un corpus del *Diari Oficial de la Generalitat de Catalunya*.

```
from nltk.corpus import PlaintextCorpusReader
from nltk import FreqDist
from nltk.util import ngrams

def score(bigram):
    score=(fdbigrams.freq(bigram)/(fdparaules.freq(bigram[0])
           *fdparaules.freq(bigram[1])))
    #score=fdbigrams[bigram]*(fdbigrams.freq(bigram)/(fdparaules.freq(bigram[0])
           *fdparaules.freq(bigram[1])))
    #score=fdbigrams[bigram]**3*(fdbigrams.freq(bigram)/(fdparaules.freq(bigram[0])
           *fdparaules.freq(bigram[1])))
    return score

reader=PlaintextCorpusReader(".","corpus-DOGC-cat.txt",encoding="utf-8")
paraules=reader.words()

bigrams=ngrams(paraules,2)

fdparaules=FreqDist(paraules)
fdbigrams=FreqDist(bigrams)

scores={}
fd=FreqDist()
for bigram in bigrams:
    if ((len(bigram[0])>3)&(len(bigram[1])>3)):
        scores[bigram]=score(bigram)
        fd[bigram]=score(bigram)
cont=0
for key in fd.keys():
```

```
    cont+=1
    bigram=key[0]+" "+key[1]
    print bigram
    if (cont==25):break
```

El programa simplement calcula els bigrames del corpus i d'aquells que estiguin compostos per paraules de més de tres lletres, calcula una puntuació i presenta les 25 col·locacions amb la puntuació més alta. En el codi es proposen tres funcions de puntuació* (*score*). En la primera definició dividim la freqüència relativa del bígram entre la multiplicació de les freqüències relatives de cada una de les paraules que el componen. Aquesta funció en dóna els millors 25 resultats següents:

***Dues d'aquestes funcions estan comentades. Per a poder provar-les simplement descomenteu la línia.**

```
Harry Potter
Pròxim Orient
Suso Lafuente
teràpies substitutives
País Basc
ginecòloga Marisa
True Star
Subirachs Torné
Ralf Schumacher
Solero Besos
Attilio Pacifico
güeroles concentrat
falange distal
Todd Field
43330 Riudoms
Grande Moreno
Artystil Iluminación
crearà distorsions
Benjamin Castaldi
Àcids nucleics
Bandi Amir
valorats discrecionalment
mantindrà ocupada
continuarà impartint
Ponce Baquero
```

Si observeu els resultats amb deteniment, veureu que la majoria d'aquests són molt poc freqüents en el corpus, ja que la nostra funció de puntuació no té prou en compte la freqüència de la col·locació mateixa. En la segona proposta, multipliquem la puntuació anterior per la freqüència absoluta del terme. Això fa que les col·locacions més freqüents tinguin una mica més d'importància. A continuació observeu els resultats que obtenim:

```
arts plàstiques
Assessoria Jurídica
Harry Potter
Pròxim Orient
Suso Lafuente
teràpies substitutives
País Basc
ginecòloga Marisa
True Star
Subirachs Torné
Ralf Schumacher
Solero Besos
Attilio Pacifico
güeroles concentrat
falange distal
Todd Field
43330 Riudoms
Grande Moreno
Artystil Iluminación
crearà distorsions
Benjamin Castaldi
Àcids nucleics
Bandi Amir
valorats discrecionalment
mantindrà ocupada
```

La tercera proposta de funció de puntuació el que fa és elevar al cub la freqüència absoluta de la col·locació i ens ofereix els resultats següents:

Podeu experimentar amb diferents funcions i observar els resultats que es van obtenint.

```
Direcció General
terme municipal
administracions públiques
Persona interessada
règim jurídic
Reial decret
ocupació temporal
Diari Oficial
Text refós
administratiu comú
Institut Català
Delegació Territorial
Aprovar definitivament
Subvenció PUOSC
procediment administratiu
llengua catalana
recurs contenciós
```

```
contenciós administratiu
jurisdicció contenciosa
Medi Ambient
sens perjudici
Tribunal Superior
Contenciosa Administrativa
Sala Contenciosa
aquesta Resolució
```

6.2. Extracció automàtica de terminologia

L'extracció automàtica de terminologia és una tasca que pretén extreure una llista de candidats a terme a partir d'un text o conjunt de textos.

La tasca és similar a la de descobriment de col·locacions, però en aquest cas ens interessa obtenir unitats terminològiques.

Aquesta tasca és útil per a confeccionar glossaris terminològics a partir de textos d'especialitat. Hi ha diverses tècniques i aquí comentarem la que s'acostuma a anomenar *extracció automàtica de terminologia estadística*. Aquesta tècnica, en la seva versió més bàsica, consisteix a calcular els *n*-grames d'un text i eliminar aquells *n*-grames que comencin o acabin amb una paraula considerada buida (*stopword*). Evidentment hi ha molts refinaments i millores d'aquesta tècnica però aquí presentarem un programa que du a terme aquesta tasca bàsica.

Nota

En aquest programa es treballen molts mètodes i instruccions que s'han anat presentant en aquest mòdul i en mòduls anteriors de l'assignatura. Deixem a càrrec vostre la comprensió d'aquests.

```
from nltk.util import ngrams
from nltk.corpus import PlaintextCorpusReader
from nltk import FreqDist

def sw_in_filter(ng):
    value=True;
    if ((ng[0].lower() in sw) or (ng[-1].lower()  in sw)):
        value=False
    else:
        for w in ng:
            if (not w.isalpha()):
                value=False

    return value


fd=FreqDist()
```

```
swf=open(”stop-cat.txt”,”rU”)
swr=swf.readlines()
sw=[]
for w in swr:
    w2=w.rstrip()
    sw.append(w2)
candidates={}
lector=PlaintextCorpusReader(".",”corpus-DOGC-cat.txt”)
frases=lector.sents()
for frase in frases:
    ngs=ngrams(frase,2)
    for ng in ngs:
        if (sw_in_filter(ng)):
            candidate=" ".join(ng)
            fd.inc(candidate)

for key in fd.keys():
    print str(fd[key]), key
```

A continuació podem observar els primers 15 candidats que proposa aquest programa (la xifra de davant del terme és la freqüència d'aparició):

```
79 terme municipal
35 Persona interessada
35 procediment administratiu
30 Reial decret
27 Diari Oficial
22 dos mesos
22 llengua catalana
21 Pressupost total
20 via administrativa
18 interposar recurs
17 Tribunal Superior
15 Medi Ambient
15 Seguretat Social
15 normativa vigent
14 Sala Contenciosa
```

6.3. Models del llenguatge

Els *n*-grames es poden fer servir per a generar models del llenguatge. Aquests models poden servir per a tasques diferents. Un exemple pot ser la generació aleatòria de text. En una sessió interactiva de Python escriviu el següent:

```
>>> import nltk
>>> from nltk.corpus import cess_cat
>>> paraules=cess_cat.words()
>>> text=nltk.Text(paraules[:50000])
>>> text.generate()
Building ngram index...
El Tribunal_Suprem -Fpa- TS -Fpt- ha presentat una proposició no de
llei al Parlament_de_Catalunya que insta a fer la inauguració ,
tot_i_que la companyia Peter_Deilmann . El 1964 *0* es pugui demostrar
a_través_de factures de venda d' aliments transgènics i dels drets
dels ciutadans estan satisfets amb la família *0* van utilitzar el
recinte durant 17 dies per als vehicles . L' informe de la població
amb una fotografia panoràmica per identificar els cims des_d' on es
projectaran els raigs de sol enviats amb miralls des_d' una
cinquantena de cims del voltant . La policia catalana busca per la
importància
```

La funció `generate` genera text aleatori a partir, en el nostre exemple, de les primeres 50.000 paraules del corpus `cess_cat`. Si torneu a generar text, veureu que aquest és diferent:

```
>>> text.generate()
El Tribunal_Suprem -Fpa- TS -Fpt- ha confirmat el conseller del DURSI
, han convocat una ballada de sardanes a més de 273.000 persones van
emprar els_seus serveis i captar els visitants finals ' ,
en_co\l.laboració_amb la Federació d' associacions de veïns del
municipi . Els serveis mínims serà a_partir_de les sis_de_la_tarda i
que si la mobilitat generada pel fet que *0* havien presentat una
proposició no de llei insta el Govern amb_l'_objecte_de fomentar l' ús
del vehicle per infracció a la supressió de dos policies nacionals ,
Bonifacio_Martín i Julián_Envit . El 19_de_desembre_de_1995 , el pati
del Seminari i el
```

7. Corpus anotats i analitzats

El paquet NLTK ens proporciona una sèrie de corpus anotats en diverses llengües. Ara per ara ens interessen els corpus etiquetats i també els analitzats sintàcticament (ja que d'aquests també podrem obtenir les etiquetes que necessitem per a entrenar-nos). A continuació presentem la llista:

Adreça web recomanada

Podeu veure una llista completa dels corpus anotats en diverses llengües del paquet NLTK en l'enllaç següent: *http://nltk.googlecode.com/svn/trunk/nltk_data/index.xml*.

- **Corpus etiquetats**
  - `brown`: Brown Corpus (anglès)
  - `indian`: Indian Language POS-Tagged Corpus (bangla, hindi, marathi, telugu)
  - `mac_morpho`: MacMorpho POS-Tagged Corpus (portuguès del Brasil)
  - `nps_chat`: NPS Chat Corpus
- **Corpus analitzats**
  - `alpino`: Alpino Treebank (neerlandès)
  - `cess_cat`: CESS-CAT Treebank (català)
  - `cess_esp`: CESS-ESP Treebank (castellà)
  - `floresta`: Floresta Treebank (portuguès)
  - `treebank`: Penn Treebank Corpus Sample (anglès)
  - `sinica`: Sinica Treebank Corpus Sample (xinès)

Per a llegir un d'aquests corpus podem fer el següent en una sessió de l'intèrpret interactiu:

```
>>> import nltk
>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ...]
```

Podem veure'n també un en català:

```
>>> nltk.corpus.cess_cat.tagged_words()
[('El', 'da0ms0'), ('Tribunal_Suprem', 'np0000o'), ...]
```

Com que aquest corpus també està segmentat per oracions, podem visualitzar-les amb la instrucció següent:

```
>>> print nltk.corpus.cess_cat.tagged_sents()
[[('El', 'da0ms0'), ('Tribunal_Suprem', 'np0000o'), ('-Fpa-', 'Fpa'),
('TS', 'np0000o'), ('-Fpt-', 'Fpt'), ('ha', 'vaip3s0'), ('confirmat', 'vmp00sm'),
```

```
('la', 'da0fs0'), ('condemna', 'ncfs000'), ('a', 'sps00'), ('quatre', 'dn0cp0'),
('anys', 'ncmp000'), ("d'", 'sps00'), ('inhabilitaci\xf3', 'ncfs000'),
('especial', 'aq0cs0'), ('i', 'cc'), ('una', 'di0fs0'), ('multa', 'ncfs000'),
('de', 'sps00'), ('3,6', 'Z'), ('milions', 'ncmp000'), ('de', 'sps00'),
('pessetes', 'Zm'), ('per', 'sps00'), ('a', 'sps00'),...
```

És possible indicar a l'NLTK que simplifiqui les etiquetes i utilitzi un conjunt d'etiquetes més reduït (que indiquen només la categoria gramatical); això ho podem fer de la manera següent:

```
>>> print nltk.corpus.cess_cat.tagged_sents(simplify_tags=True)
[[('El', 'D'), ('Tribunal_Suprem', 'N'), ('-Fpa-', 'F'), ('TS', 'N'),
('-Fpt-', 'F'), ('ha', 'V'), ('confirmat', 'V'), ('la', 'D'),
('condemna', 'N'), ('a', 'S'), ('quatre', 'D'), ('anys', 'N'), ("d'", 'S'),
('inhabilitaci\xf3', 'N'), ('especial', 'A'), ('i', 'C'), ('una', 'D'),
('multa', 'N'), ('de', 'S'), ('3,6', 'Z'), ('milions', 'N'),
('de', 'S'), ('pessetes', 'Z'), ('per', 'S'), ('a', 'S'),...
```

De fet aquest corpus està analitzat i per tant podem també visualitzar-lo de la manera següent:

```
>>> print nltk.corpus.cess_cat.parsed_sents()
[Tree('S', [Tree('sno-SUJ', [Tree('espec.ms', [Tree('da0ms0', ['El'])]),
Tree('grup.nom.ms', [Tree('np0000o', ['Tribunal_Suprem']), Tree('sno',
[Tree('grup.nom.ms', [Tree('Fpa', ['-Fpa-']), Tree('np0000o', ['TS']),
Tree('Fpt', ['-Fpt-'])])])])]), Tree('grup.verb', [Tree('vaip3s0', ['ha']),
Tree('vmp00sm', ['confirmat'])]), Tree('sn-CD', [Tree('espec.fs',
[Tree('da0fs0', ['la'])]), Tree('grup.nom.fs', [Tree('ncfs000', ['condemna']),
Tree('sp', [Tree('prep', [Tree('sps00', ['a'])]),...
```

8. Recursos lèxics de l'NLTK: WordNet

NLTK proporciona una sèrie de recursos lèxics, entre els quals es pot destacar el WordNet.

WordNet (Miller, 1990) és una base de dades lèxica de l'anglès. Aquesta base de dades conté noms, verbs, adjectius i adverbis que s'agrupen en conjunts de sinònims cognitius (*synset*). Cada un d'aquests *synsets* expressa un concepte.

Adreça web recomanada

Wordnet es pot consultar i descarregar a *http://wordnet.princeton.edu/*

En aquesta base de dades els *synsets* estan enllaçats mitjançant relacions semàntiques.

NLTK proporciona accés al WordNet, i aprofitarem aquest accés per a conèixer una mica més l'estructura d'aquesta base de dades lèxica.

Els exemples d'aquest apartat estan estrets de Bird i altres (2009).

8.1. Sentits i sinònims

En anglès, les paraules *motorcar* i *automobile* són sinònimes. Vegem ara com podem obtenir aquesta informació amb el WordNet i NLTK:

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

Veiem que l'únic *synset* associat a aquesta paraula és el sentit 1 del substantiu *car*. Podem consultar també quines paraules comparteixen aquest *synset*.

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

També podem consultar la definició del sentit 1 del substantiu *car* i fins i tot exemples d'ús.

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

Tot i que les definicions ens poden resultar d'utilitat per a entendre el significat d'un determinat *synset*, per als nostres programes seran més útils les paraules del *synset*. Per a eliminar ambigüitats les paraules s'identifiquen amb el parell *synset*-paraula, conjunt que anomenarem *lema**.

***Per exemple, "car.n01.automobile".**

```
>>> wordnet.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')]
>>> wordnet.lemma('car.n.01.automobile')
Lemma('car.n.01.automobile')
>>> wordnet.lemma('car.n.01.automobile').synset
Synset('car.n.01')
>>> wordnet.lemma('car.n.01.automobile').name
'automobile'
```

Les paraules *automobile* i *motorcar* no són ambigües i per aquest motiu tenen associat un únic *synset*. En canvi la paraula *car* sí que és ambigua i té associada diversos *synsets*:

```
>>> wordnet.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 wordnet.synsets('car'):
...     print synset.lemma_names, synset.definition
...
['car', 'auto', 'automobile', 'machine', 'motorcar'] a motor vehicle with four wheels;
usually propelled by an internal combustion engine
['car', 'railcar', 'railway_car', 'railroad_car'] a wheeled vehicle adapted to the
rails of railroad ['car', 'gondola'] the compartment that is suspended from an airship
and that carries personnel and the cargo and the power plant
['car', 'elevator_car'] where passengers ride up and down
['cable_car', 'car'] a conveyance for passengers or freight on a cable railway
```

També podem accedir a tots els lemes de la paraula *car* de la manera següent:

```
>>> wordnet.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')]
```

8.2. La jerarquia de WordNet

Els *synsets* de WordNet corresponen a conceptes abstractes que no sempre tenen paraules associades. Aquests conceptes estan organitzats en una jerarquia. Alguns conceptes són molt generals i d'altres són més específics. És fàcil navegar per aquesta jerarquia. Per exemple, amb un concepte com *motorcar*, podem consultar els *hipònims* immediats, és a dir, els conceptes que són més específics.

```
>>> motorcar=wordnet.synset('car.n.01')
>>> tipus_de_motorcar=motorcar.hyponyms()
>>> sorted([lemma.name for synset in tipus_de_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']
```

També podem navegar per la jerarquia i consultar els *hiperònims*, és a dir, els conceptes menys específics. Algunes paraules tenen diversos camins, atès que es poden classificar de més d'una manera. Hi ha dos camins entre "car.n.01" i "entity.n.01" perquè "wheeled_vehicle.n.01" es pot classificar tant com a vehicle com a contenidor.

```
>>> 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']
```

Podem obtenir l'hiperònim més general (o hiperònim arrel) d'un *synset* de la manera següent:

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

8.3. Altres relacions lèxiques

Les relacions d'hiperonímia i hiponímia són relacions lèxiques perquè relacionen *synsets*. Aquesta relació es basa en la propietat *és un*. Una altra relació possible és la que hi ha entre un objecte i els seus components (*meronímia* i *holonímia*). Un merònim és una paraula el significat de la qual constitueix una part del significat d'una altra, que s'anomena *holònim*. També es pot parlar de meronímia quant a la substància de què es compon i d'holonímia quant al fet que és membre de... Per exemple:

```
>>> wordnet.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')]
>>> wordnet.synset('tree.n.01').substance_meronyms()
[Synset('heartwood.n.01'), Synset('sapwood.n.01')]
>>> wordnet.synset('tree.n.01').member_holonyms()
[Synset('forest.n.01')]
```

També es poden establir relacions entre verbs. Per exemple, l'acció de *walking* implica la de *stepping*, de manera que *walking* implica (*entails*) *stepping*. Alguns verbs tenen diverses implicacions (*entailments*):

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

L'*antonímia* és una relació semàntica entre lemes:

```
>>> wordnet.lemma('supply.n.02.supply').antonyms()
[Lemma('demand.n.02.demand')]
>>> wordnet.lemma('rush.v.01.rush').antonyms()
[Lemma('linger.v.04.linger')]
>>> wordnet.lemma('horizontal.a.01.horizontal').antonyms()
```

```
[Lemma('vertical.a.01.vertical'), Lemma('inclined.a.02.inclined')]
>>> wordnet.lemma('vertical.a.01.vertical').antonyms()
[Lemma('horizontal.a.01.horizontal'), Lemma('inclined.a.02.inclined')]
```

8.4. Similaritat semàntica

WordNet ens ofereix una extensa i complexa xarxa semàntica. Si tenim un *synset* determinat podem trobar altres *synsets* que tinguin un significat similar. Conèixer les paraules relacionades semànticament pot ser d'utilitat per a diferents tasques. Per exemple, si volem cercar textos amb un terme general podem trobar termes més específics i cercar textos que continguin aquests termes específics.

Si ens fixem, cada *synset* té un o més camins d'hiperonímia que connecten amb un hiperònim arrel (per exemple "entity.n.01"). Podem quantificar el concepte de generalitat cercant la profunditat de cada *synset* en la jerarquia.

```
>>> wordnet.synset('baleen_whale.n.01').min_depth()
14
>>> wordnet.synset('whale.n.01').min_depth()
5
>>> wordnet.synset('vertebrate.n.01').min_depth()
8
>>> wordnet.synset('entity.n.01').min_depth()
0
```

Podem establir una mesura de similaritat lèxica fixant-nos en els camins del *synset* fins a l'hiperònim arrel respecte a un altre *synset*.

```
>>> wordnet.synset('car.n.01').path_similarity(wordnet.synset('car.n.01'))
1.0
>>> wordnet.synset('car.n.01').path_similarity(wordnet.synset('motorcycle.n.01'))
0.33333333333333331
>>> wordnet.synset('car.n.01').path_similarity(wordnet.synset('bicycle.n.01'))
0.20000000000000001
>>> wordnet.synset('car.n.01').path_similarity(wordnet.synset('shoe.n.01'))
0.10000000000000001
>>> wordnet.synset('car.n.01').path_similarity(wordnet.synset('elephant.n.01'))
0.058823529411764705
```

Podem explorar totes les relacions d'un *synset* fent:

```
>>> dir(wordnet.synset('harmony.n.02'))
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__getattribute__',
'__hash__', '__init__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__str__', '__weakref__',
'_iter_hypernym_lists', '_lemma_pointers', '_pointers', '_related',
'_wordnet_corpus_reader', 'also_sees', 'attributes', 'causes', 'closure',
'common_hypernyms', 'definition', 'entailments', 'examples', 'frame_ids',
'hypernym_distances', 'hypernym_paths', 'hypernyms', 'hyponyms',
'instance_hypernyms', 'instance_hyponyms', 'jcn_similarity',
'lch_similarity', 'lemma_infos', 'lemma_names', 'lemmas', 'lexname',
'lin_similarity', 'lowest_common_hypernyms', 'max_depth', 'member_holonyms',
'member_meronyms', 'min_depth', 'name', 'offset', 'part_holonyms', 'part_meronyms',
'path_similarity', 'pos', 'res_similarity', 'root_hypernyms',
'shortest_path_distance', 'similar_tos', 'substance_holonyms',
'substance_meronyms', 'tree', 'verb_groups', 'wup_similarity']
```

8.5. El WordNet català

S'han creat WordNets en altres llengües, com el català (Benitez i altres, 1998). Treballarem amb una petita part del WordNet català, la que es distribueix amb l'analitzador Freeling. L'arxiu el teniu disponible entre els programes d'aquest mòdul i s'anomena `wordnet-freeling-cat.txt`. L'arxiu té l'estructura següent:

```
S:00001740:A capaç
S:00001740:N entitat
S:00001740:V respirar
S:00002062:A incapaç
S:00002086:N organisme ésser
S:00002484:V aspirar
S:00002880:N vida
S:00003011:V sospirar
S:00003057:A naixent
S:00003095:N cè\l.lula
S:00003469:A incipient naixent
S:00003731:N causa
S:00003763:V aspirar
S:00003965:A últim
S:00004123:N individu persona ànima
S:00004364:A últim
S:00004816:A tallat
S:00005052:V bufar
S:00005288:A complet
```

```
S:00005316:V olorar
...
W:Adam:N 06928214
W:abaixar:V 01343131 01228705 00182537 01411253 00382538
W:abandonament:N 00030682 00134056 03677676 05601181 00134849 00056297 00139061
03835201
W:abandonar:V 01376117 00346044 01623741 00415625 01524047 01609431 01524319
00734233 01421290 01525019 00415168 00253929 01760985 01761339 00415041 01728889
00345904 01521543
W:abandonat:A 00996592 01192009
W:abandó:N 00030682 00134056 05601181 03677676 00134849 00139061 00056297 03835201
W:abast:N 03993027 10045533 06357619 00684444 00218788 04354914
W:abastar:V 01689594 00818616
W:abatre:V 00052062 01238101 00734516
W:abella:N 01710960
W:abisme:N 06735354 06327598 03734192 06735286
W:abocar:V 01064888 01352959 01416151 01416698 01061371 00313984 00943569 01060886
01346440 01346751 01416816
W:abonar:V 00492800
W:abordar:V 00531339
W:abraçada:N 00268156 00033679 00268334
W:abraçar:V 00976586 01832499 00976384 00406673
W:abric:N 03315837 03316185
W:abril:N 10923938
W:absent:A 01775263 00166902
W:absolut:A 00005515 00489625 00523702 01844500 00684153 02095346 00490201
W:absorbir:V 00405557 01059371 01086552 00147396 00274622 00318860
```

Com podem observar l'estructura és doble: per una banda hi ha unes entrades que comencen per S (*synset*) i l'estructura és el número identificador del *synset*, la categoria gramatical i la paraula associada. Les entrades que comencen per W (*word*) tenen una estructura diferent: en primer lloc la paraula, a continuació la categoria i per finalitzar els diferents números de *synsets* associats a la paraula.

A l'arxiu `wordnet-freeling-eng.txt` també podem veure el corresponent WordNet en anglès, que té una estructura idèntica.

```
W:a-ok:J 01075873
W:a-okay:J 01075873
W:a.m.:N 10937566
W:aardvark:N 01593926
W:aardwolf:N 01627385
W:aba:N 02155402 02155313
W:abaca:N 10725396 08656674
W:aback:R 00073386 00073303
```

```
W:abactinal:J 01606838
W:abacus:N 02155652 02155519
W:abalone:N 01455925
W:abamp:N 09802897
W:abampere:N 09802897
W:abandon:N 03826829 05561743
W:abandon:V 01524319 01524047 01421290 00415168 00415625
W:abandoned:J 01258647 00996592 01262162
W:abandonment:N 00134056 00030682 00056297
W:abarticulation:N 10263375
W:abase:V 01228249
W:abasement:N 10360885 00175673
...
S:00001740:J able
S:00001740:N entity something
S:00001740:R ahorse ahorseback
S:00001740:V breathe respire
S:00001874:R just scarce
S:00002062:J unable
S:00002086:N being organism
S:00002143:V choke
S:00002287:J abaxial dorsal
S:00002343:V hyperventilate
S:00002477:R basically
S:00002493:J adaxial ventral
S:00002561:V belch bubble burp eruct
S:00002687:J abducent abducting
S:00002841:V hiccup
S:00002856:J adducent adducting adductive
S:00002869:R boiling
S:00002880:N life
S:00002940:R enviably
S:00003011:N biont
S:00003011:V sigh
```

També s'ha inclòs l'estructura del WordNet a l'arxiu `wn16.src`:

```
00001740:A - 00 -
00001740:N - 03 1stOrderEntity
00001740:R - 02 -
00001740:V - 29 SecondOrderEntity
00001874:R - 02 -
00002062:A - 00 -
00002086:N 00001740 03 Living
00002143:V 00001740 29 SecondOrderEntity
00002191:R - 02 -
```

```
00002287:A - 00 -
00002343:V 00001740 29 SecondOrderEntity
00002374:R - 02 -
00002477:R - 02 -
00002484:V 00003763 29 SecondOrderEntity
00002493:A - 00 -
00002561:V 00071495 29 SecondOrderEntity
00002687:A - 00 -
00002720:V 00071495 29 SecondOrderEntity
00002794:R - 02 -
00002841:V 00001740 29 SecondOrderEntity
```

Aquesta estructura és més complexa i no entrarem en detalls en aquest apartat.

Podem processar aquests arxius per a fer diverses tasques. El que farem per a demostrar alguna aplicació és desenvolupar un diccionari anglès-català aprofitant els WordNets en anglès i català (`creardiccionary.py`).

```
import codecs
eng=open("wordnet-freeling-eng.txt","rU")
dicteng={}
for linia in eng:
    if linia.startswith("S"):
        camps1=linia.rstrip().split(" ")
        camps2=camps1[0].split(":")
        clau=camps2[1]+":"+camps2[2]
        valor=", ".join(camps1[1:])
        dicteng[clau]=valor


cat=codecs.open("wordnet-freeling-cat.txt","rU","ISO-8859-1")
for linia in cat:
    if linia.startswith("W"):
        camps1=linia.rstrip().split(" ")
        camps2=camps1[0].split(":")
        paraula=camps2[1]
        categoria=camps2[2]
        synsets=camps1[1:]
        for synset in synsets:
            synsetcat=synset+":"+categoria
            if dicteng.has_key(synsetcat):
                print paraula+"("+categoria+"): "+dicteng[synsetcat]
```

Exercici

Modifica el programa anterior perquè doni la sortida a un fitxer i no escrigui entrades repetides.

Resum

En aquest mòdul hem presentat els conceptes bàsics relacionats amb l'anàlisi de corpus. Hem presentat en detall la codificació de caràcters, aspecte molt important per a poder tractar de manera correcta corpus textuals i evitar problemes amb els caràcters.

També hem presentat a fons dues tasques bàsiques, la segmentació en unitats lèxiques (*tokenització*) i la segmentació en oracions. Aquestes tasques són bàsiques en el processament del llenguatge natural i han estat àmpliament estudiades. A part de les tècniques bàsiques emprades els resultats d'aquestes tasques es poden millorar afegint coneixement lèxic en forma de diccionaris (Grefenstette i Tapanainen, 1994). També hi ha propostes que apliquen tècniques d'aprenentatge per a induir regles de segmentació a partir d'una sèrie de *tokens* ambigus distribuïts en un document (Mikheev, 2002).

Bibliografia

Benitez, L.; Cervell, S.; Escudero, G.; Lopez, M.; Rigau, G. i Taule, M. (1998). «Methods and Tools for Building the Catalan WordNet». A: «In Proceedings of ELRA Workshop on Language Resources for European Minority Languages», URL: http://www.citebase.org/abstract?id=oai:arXiv.org:cmp-lg/9806009.

Bird, S.; Ewan, K. i Edwar, L. (2009). *Natural Language Processing with Python - Analyzing Text with the Natural Language Toolkit*. O'Reilly Media.

Boleda, G.; Bott, S.; Poblete, B.; Castillo, C.; Fuenmayor, M.; Badia, T. i López, V. (2004). «CucWeb: un corpus del català contruït a partir de la Web». A: «II congrés online de l'Observatori per a la Cibersocietat».

Grefenstette, G. i Tapanainen, P. (1994). «What is a word, What is a sentence? Problems of Tokenization». Report tècnic, Xerox.

Kilgarriff, A. i Grefenstette, G. (2003). «Introduction to the Special Issue on the Web as Corpus». A: *Computational Linguistics*, volum 29(3): pàgs. 333–347. doi: 10.1162/089120103322711569. http://www.mitpressjournals.org/doi/pdf/10.1162/089120103322711569, URL: http://www.mitpressjournals.org/doi/abs/10.1162/089120103322711569.

Mikheev, A. (2002). «Periods, capitalized words, etc». A: *Computational Linguistics*, volum 28(3): pàgs. 239–318.

Miller, G. (1990). «WordNet: an on-line lexical database». A: *International Journal of Lexicography*, volum 3(4).

Palmer, D. (2000). *Handbook of Natural Language Processing*, capítol 2. Tokenisation and sentence segmentation, (pàgs. 11–36). New York: Marcel Dekker, Inc. ISBN 0824790006.

Zipf, G. (1949). *Human Behaviour and the Principle of Least Effort*. Cambridge: Addison-Wesley.