Anàlisi fragmental (*chunking*)

Antoni Oliver

PID_00155234

Universitat Oberta de Catalunya
www.uoc.edu

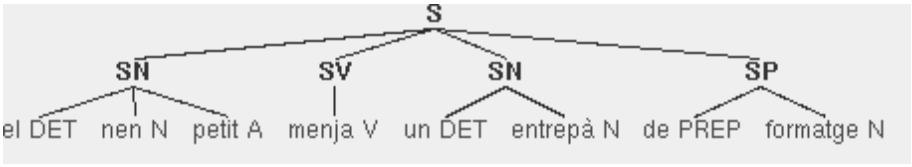
Índex

Introducció

En el mòdul "Etiquetatge morfosintàctic" vam veure el nivell d'anàlisi morfosintàctica que consisteix a donar a cada paraula d'un text una etiqueta amb informació morfosintàctica. En aquest mòdul veurem un nivell d'anàlisi superior que consisteix a agrupar conjunts de paraules d'un text en uns grups amb un cert criteri sintàctic. Bàsicament l'anàlisi fragmental o *chunking* consisteix a determinar els constituents sintàctics d'una oració, tot i que no estableix la relació entre aquests constituents.

En la figura 1 podeu observar una anàlisi fragmental de l'oració "El nen petit menja un entrepà de formatge".

Figura 1. Anàlisi fragmental



Com podeu observar, l'anàlisi de constituents (sintagmes nominals, verbals i preposicionals) és correcta, però no s'estableix una relació adequada entre aquests constituents. Tots els sintagmes pengen de l'oració.

Objectius

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

1. Saber què és l'anàlisi fragmental (*chunking*).
2. Saber crear *chunkers* senzills amb Python i NLTK.
3. Conèixer algunes aplicacions de l'anàlisi fragmental.

1. El concepte de *chunk*, anàlisi fragmental i anàlisi sintàctica superficial

Els conceptes de *chunk*, anàlisi fragmental (*chunking*) i anàlisi sintàctica superficial (*shallow parsing*) estan relacionats i sovint es confonen.

El concepte de *chunk* és controvertit i sovint es defineix més aviat des d'un punt de vista operacional que des d'un punt de vista lingüístic.

> El terme *chunk* va ser emprat per primera vegada per Abney (1996), que el defineix com "el fragment no recursiu d'un constituent d'una oració simple (*clause*) que abraça des del començament del constituent fins al seu nucli (*head*) però sense incloure els possibles postmodificadors d'aquest nucli".

Altres definicions identifiquen un *chunk* amb un sintagma no recursiu corresponent a una categoria lèxica principal (nom, adjectiu, preposició i verb), i admeten que junt amb el nucli poden incloure tant premodificadors com postmodificadors. Totes les definicions coincideixen a incloure en un *chunk* un element nuclear bàsic i obligatori que pot estar acompanyat de modificadors no recursius (és a dir, els modificadors no poden incloure ni directament ni indirectament elements de la mateixa categoria que el nucli).

> **Exemple**
>
> Si agafem com a exemple *el vell gat rabiüt és molt ferotge*, un *chunk* nominal seria *el vell gat rabiüt*, ja que inclou un nucli nominal (aquí un nom) complementat amb tres modificadors (un determinant i dos adjectius). En canvi, en l'oració, *el gat del veí és molt ferotge*, el *chunk* nominal és *el gat*, ja que el modificador és recursiu perquè conté un altre nom. També, en l'oració *el gat que vaig veure ahir és molt ferotge* el *chunk* nominal és *el gat*, ja que el modificador conté una altra oració.

Molt sovint el concepte de *chunk* es redueix al d'un sintagma nominal bàsic o no recursiu (base NP), ja que moltes aplicacions estan simplement interessades en la localització o extracció d'aquestes unitats. Altres sistemes estenen la definició a qualsevol categoria sintàctica. Per exemple, Tjong Kim Sang i Bucholz (2000), descriuen la tasca de *chunking* per a la competició portada a terme en el CoNLL-2000. La tasca que proposen consisteix a dividir el text en grups no superposats de paraules relacionades sintàcticament. És a dir, el *chunkeig* consisteix a dividir el text en grups de manera que les paraules relacionades sintàcticament siguin membres d'un mateix grup. Aquests grups no poden estar superposats, és a dir, una determinada paraula només pot ser membre d'un grup. Més endavant en aquest mateix mòdul podrem treballar amb el corpus *chunkejat* que es va crear per a aquesta competició. Aquest corpus s'ha creat a partir d'un *treebank* és a dir, a partir d'un corpus analitzat sintàcticament, concretament a partir de la secció corresponent al *Wall Street Journal* (WSJ) del

corpus Penn Treebank II. En el *treebank* es distingeixen els tipus de sintagmes següents:

- NP: noun phrase
- VP: verb phrase
- PP: prepositional phrase
- ADVP: adverb phrase
- SBAR: subordinate clause
- ASDJP: adjective phrase
- PRT: particles
- CONJP: conjunction phrase
- INTJ: interjection
- LST: list marker
- UCP: unlike coordinated phrase

Els tipus de *chunks* que es defineixen són els mateixos, però com que els *chunks* han de ser grups no superposats, això fa que hi hagi una sèrie de particularitats:

- **NP:** *chunks* de tipus nominal

- **VP:** *chunks* de tipus verbal

- **ADVP:** *chunks* de tipus adverbial

```
(NP We/PRP)
(ADVP still/RB)
(VP have/VBP)
(NP people/NNS)
(VP wandering/VBG)
(ADVP around/IN)
(PP in/IN)
(NP a/DT daze/NN)
(PP in/IN)
(NP San/NNP Francisco/NNP)
(VP worrying/VBG)
(PP about/IN)
(SBAR whether/IN)
(NP it/PRP)
(VP 's/VBZ going/VBG to/TO rain/VB)
(NP tonight/RB)
```

- **ADJP:** *chunks* de tipus adjectival

```
(NP The/DT prisons/NNS)
(VP are/VBP)
(ADJP too/RB crowded/VBN)
```

- **PP:** *chunks* de tipus preposicional, la majoria constituïts per una única paraula, la preposició

```
But/CC
(NP other/JJ Bush/NNP administration/NN officials/NNS)
(VP have/VBP criticized/VBN)
(NP Maryland/NNP Gov./NNP William/NNP Schaefer/NNP)
(PP for/IN)
(VP blocking/VBG)
(NP the/DT use/NN)
(PP of/IN)
(NP possible/JJ sites/NNS)
(PP in/IN)
(NP that/DT state/NN)
```

- **SBAR:** el *chunk* normalment consisteix en una única paraula, el complementador

```
(NP What/WP)
(SBAR if/IN)
(NP it/PRP)
(VP happened/VBD)
(PP to/TO)
(NP us/PRP)
```

- **CONJP:** en el *treebank* es marquen amb aquesta etiqueta únicament les conjuncions formades per més d'una paraula, i per tant els *chunks* d'aquest tipus estaran formats per conjuncions de més d'una paraula.

```
(NP Newport/NNP Beach/NNP operations/NNS)
(VP differ/VBP)
(PP from/IN)
(NP the/DT Hollywood/NNP boiler/NN rooms/NNS)
(PP in/IN)
(NP style/NN)
(CONJP as/RB well/RB as/IN)
(PP in/IN)
(NP dollars/NNS)
```

- **PTR:** es marquen partícules verbals, en generals formades per una única paraula excepte *on and off*.

- **INTJ:** és un *chunk* molt poc freqüent format per una interjecció.

```
(INTJ Please/VB)
(VP excuse/VB)
(NP my/PRP$ handwriting/NN)
```

- **LST:** són *chunks* poc freqüents que es fan servir en llistes.

```
(LST 1/CD)
./.
(VP Make/VB)
(ADJP sure/JJ)
(NP you/PRP)
(VP have/VBP)
(NP a/DT strong/JJ personnel/NNS department/NN)
```

- **UCP:** és un *chunk* conseqüència de la mateixa etiqueta del *treebank*.

```
(VP will/MD generate/VB)
(NP enough/JJ lawsuits/NNS)
(VP to/TO keep/VB)
(NP this/DT city/NN)
(NP 's/POS)
(UCP personal-injury/NN and/CC construction/NN)
(NP lawyers/NNS)
(ADJP busy/JJ)
(PP for/IN)
(NP quite/RB some/DT time/NN)
```

Com a conclusió, l'**anàlisi fragmental** (*chunking*) té com a finalitat localitzar fragments de text que puguin ser analitzats sense pretendre analitzar la totalitat del text ni establir les dependències sintàctiques entre els fragments analitzats.

L'**anàlisi superficial** (*shallow parsing*), en canvi, és una anàlisi sintàctica d'un text que reconeix únicament algunes estructures sintàctiques concretes, com per exemple el verb i el seu objecte directe, o una oració preposicional, però sense indicar de quin element depèn.

Molt sovint, l'anàlisi sintàctica superficial consisteix a detectar els nuclis dels sintagmes i oracions del text i les dependències entre si, però sense construir l'arbre d'anàlisi. El resultat se sol expressar en forma de graf de dependències, i normalment no resol dependències a llarga distància o entre nuclis de profunditats molt diferents dins de l'arbre d'anàlisi. És per aquest motiu que s'anomena *anàlisi superficial*. Un analitzador superficial generalment està format per tres mòduls (Hammerton i altres, 2002):

- Etiquetatge morfosintàctic
- *Chunking*
- Establiment de la relació entre *chunks*

2. Un *chunker* senzill

2.1. Primera versió del *chunker*

Amb NLTK és senzill crear *chunkers*. En el nostre primer exemple en podem veure un capaç de detectar sintagmes nominals. El *chunker* necessitarà tenir l'oració analitzada morfosintàcticament i en aquesta versió proporcionarem aquesta anàlisi directament (`programa-6-1.py`).

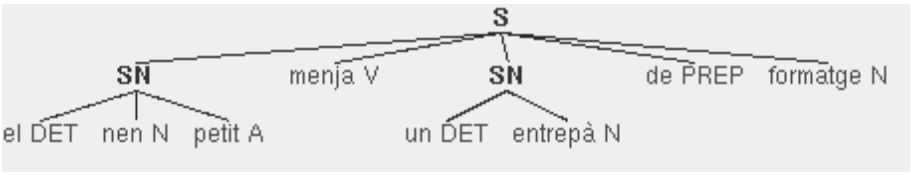
```
#! --*-- encoding=utf8 --*--
import nltk
from nltk.parse import Tree
#per a poder chunkejar necessitem tenir la frase analitzada
analisi=[("el", "DET"), ("nen", "N"), ("petit", "A"), ("menja", "V"),
 (“un", "DET"), ("entrepà" ,"N"), ("de" ,"PREP"), ("formatge","N")]

#definim una gramàtica
gramatica = "SN: {<DET><N><A>?}"

#definim un chunker per expressions regulars
cp=nltk.RegexpParser(gramatica)

#determinem els chunks de l'oració
chunks=cp.parse(analisi)

#escrivim per pantalla els chunks
print chunks

#dibuixem els chunks
chunks.draw()
```

Aquest programa ens oferirà per pantalla la representació següent:

```
(S
  (SN el/DET nen/N petit/A)
  menja/V
  (SN un/DET entrepà/N)
  de/PREP
  formatge/N)
```

I a més dibuixarà l'estructura arbòria que podem observar a la figura 2.

Figura 2. Anàlisi fragmental (incompleta)



Si volem que l'anàlisi fragmental sigui més completa haurem de definir també les expressions regulars corresponents en el sintagma verbal i en el sintagma preposicional (`programa-6-2.py`).

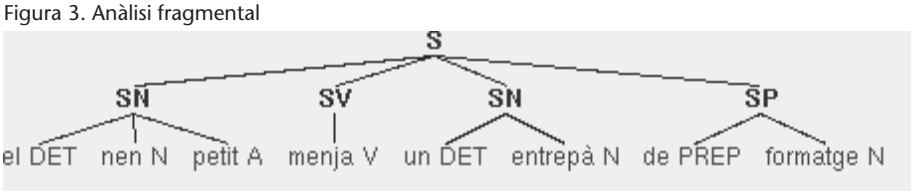
```
#! --*-- encoding=utf8 --*--
import nltk
from nltk.parse import Tree
#per a poder chunkejar necessitem tenir la frase analitzada
analisi=[("el", "DET"), ("nen", "N"), ("petit", "A"), ("menja", "V"),
    (“un", "DET"), ("entrepà" ,"N"), ("de" ,"PREP"), ("formatge","N")]

#definim una gramàtica
gramatica = """
SN: {<DET><N><A>?}
SV: {<V>}
SP: {<PREP><N><A>?}
"""

#definim un chunker per expressions regulars
cp=nltk.RegexpParser(gramatica)

#determinem els chunks de l'oració
chunks=cp.parse(analisi)

#escrivim per pantalla els chunks
print chunks

#dibuixem els chunks
chunks.draw()
```

Ara la sortida del programa és:

```
(S
  (SN el/DET nen/N petit/A)
  (SV menja/V)
  (SN un/DET entrepà/N)
  (SP de/PREP formatge/N))
```

I la representació arbòria és la que podem observar a la figura 3.

Figura 3. Anàlisi fragmental



2.2. Ús d'un etiquetador morfosintàctic

Sabem crear i emmagatzemar etiquetadors morfosintàctics. Ara recuperararem un dels etiquetadors i el farem servir en el nostre *chunker* (`programa-6-3.py`).

Vam aprendre a crear i emmagatzemar etiquetadors morfosintàctics en el mòdul "Etiquetatge morfosintàctic".

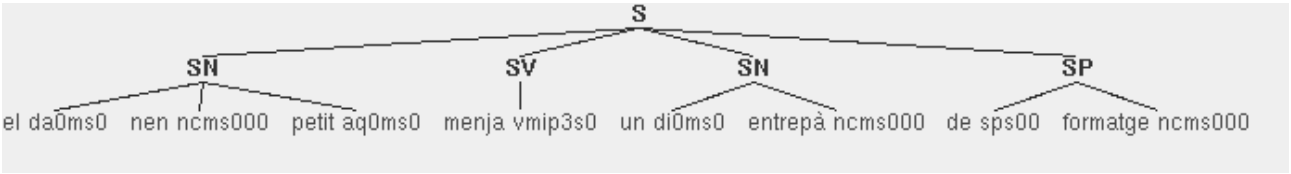
```
#! --*-- encoding=utf8 --*--
import nltk
from nltk import chunk
from nltk.parse import Tree
from cPickle import load

entrada=open('etiquetador.pkl','rb')
etiquetador=load(entrada)

frase="el nen petit menja un entrepà de formatge"
tokens=frase.split(" ")
tokenana=etiquetador.tag(tokens)

print tokenana

#definim una gramàtica
gramatica = """
SN: {<d.*><n.*><a.*>?}
SV: {<v.*>}
SP: {<sp.*><n.*><a.*>?}
"""

#definim un chunker per expressions regulars
cp=nltk.RegexpParser(gramatica)

#determinem els chunks de l'oració
chunks=cp.parse(tokenana)

#escrivim per pantalla els chunks
print chunks

#dibuixem els chunks
chunks.draw()
```

Aquest programa ens imprimeix per pantalla tant l'anàlisi morfosintàctica (`print tokenana`) com els *chunks* (`print chunks`):

```
[('el', 'da0ms0'), ('nen', 'ncms000'), ('petit', 'aq0ms0'), ('menja', 'vmip3s0'),
('un', 'di0ms0'), ('entrep\xc3\xa0', 'ncms000'), ('de', 'sps00'),
('formatge', 'ncms000')]
(S
  (SN el/da0ms0 nen/ncms000 petit/aq0ms0)
  (SV menja/vmip3s0)
  (SN un/di0ms0 entrepà/ncms000)
  (SP de/sps00 formatge/ncms000))
```

També ens dibuixa l'anàlisi fragmental de l'oració (figura 4).

Figura 4. Anàlisi fragmental



Fixeu-vos que hem canviat les expressions regular que defineixen la gramàtica per a adaptar-les a les etiquetes que proporciona el nostre analitzador:

```
gramatica = """
SN: {<d.*><n.*><a.*>?}
SV: {<v.*>}
SP: {<sp.*><n.*><a.*>?}
"""
```

En aquest cas l'ús de `.*` significa qualsevol caràcter, així `<n.*>` significa qualsevol etiqueta que comenci per `n`.

3. Chinking

Per ara hem definit els nostres *chunkers* a partir dels elements que constitueixen el *chunk*. En algunes situacions és més senzill definir el que volem excloure d'un *chunk*. Es pot definir un *chink* com un fragment del text que no és un *chunk*. El que s'acostuma a fer per a *chinkejar* és crear un *chunk* que agafi tot el text i després definir expressions regulars que excloguin certs fragments. Fixem-nos en l'exemple següent (`programa-6-4.py`):

```
#! --*-- encoding=utf8 --*--
import nltk
from nltk import chunk
from nltk.parse import Tree
from cPickle import load

entrada=open('etiquetador.pkl','rb')
etiquetador=load(entrada)

frase="el nen petit menja a l'escola"
tokens=frase.split(" ")
tokenana=etiquetador.tag(tokens)

print tokenana

#definim una gramàtica
gramatica = """
SN:
    {<.*>+}     #ho chunkeja tot
    }<v.*|sp.*>{ #exclou seqüències de verbs i preposicions
"""

#definim un chunker per expressions regulars
cp=nltk.RegexpParser(gramatica)

#determinem els chunks de l'oració
chunks=cp.parse(tokenana)

#escrivim per pantalla els chunks
print chunks

#dibuixem els chunks
chunks.draw().
```

La gramàtica és capaç de detectar els SN de l'oració. La primera expressió *chunkeja* tota l'oració com a SN i la segona elimina les seqüències de verbs i preposicions, i divideix l'oració en dos sintagmes nominals. Fixem-nos que els *chinks* els hem creat amb } i {.

4. Representació de *chunks*: etiquetes i arbres

Hi ha diverses maneres d'expressar *chunks*. Fins ara hem vist la notació d'arbre, com per exemple:

```
(S
  (SN el/DET nen/N petit/A)
  (SV menja/V)
  (SN un/DET entrepà/N)
  (SP de/PREP formatge/N))
```

Hi ha una altra notació, denominada format IOB, que té l'aspecte següent:

```
el DET B-SN
nen N I-SN
petit A I-SN
menja V B-SV
un DET B-SN
entrepà N I-SN
de SP B-SP
formatge N I-SP
```

Fixeu-vos que en primer lloc es presenta la paraula, després l'etiqueta corresponent a aquesta paraula i finalment una etiqueta relativa a *chunks*. Aquesta tercera etiqueta comença per B (inici o frontera) i per I (interior). Per exemple, l'etiqueta B-NP indica que és un inici de NP i I-NP indica interior de NP.

El programa següent (`programa-6-5.py`) mostra tot el contingut de la part d'entrenament del corpus CoNll2000 en aquest format:

```
import nltk
train=nltk.corpus.conll2000.chunked_sents('train.txt')
for t in train:
        for word, tag, chtag in nltk.chunk.tree2conlltags(t):
            print word+" "+tag+" "+chtag
```

La sortida d'aquest programa és una cosa de l'estil:

```
Alusuisse NNP B-NP
of IN B-PP
America NNP B-NP
Inc. NNP I-NP
plans VBZ B-VP
to TO I-VP
sell VB I-VP
its PRP$ B-NP
Consolidated NNP I-NP
Aluminum NNP I-NP
Corp. NNP I-NP
```

Fixeu-vos que el programa fa servir la funció `tree2conlltags`, que transforma un arbre en notació IOB. També és possible seleccionar un tipus de *chunk* determinat, com en l'exemple següent (`programa-6-5b.py`):

```
import nltk
train=nltk.corpus.conll2000.chunked_sents('train.txt', chunk_types=['NP'])
for t in train:
        for word, tag, chtag in nltk.chunk.tree2conlltags(t):
            print word+" "+tag+" "+chtag
```

I la sortida és:

```
Alusuisse NNP B-NP
of IN O
America NNP B-NP
Inc. NNP I-NP
plans VBZ O
to TO O
sell VB O
its PRP$ B-NP
Consolidated NNP I-NP
Aluminum NNP I-NP
Corp. NNP I-NP
```

Fixeu-vos que com que hem seleccionat només els *chunks* de tipus NP tot el que estigui fora d'aquest tipus de *chunk* rep l'etiqueta O.

5. Creació i avaluació d'un *chunker* a partir de corpus

En el mòdul "Etiquetatge morfosintàctic" vam veure que és possible entrenar un etiquetador a partir d'un corpus etiquetat. Ara veurem que també és possible crear *chunkers* a partir de corpus que estiguin *chunkejats*. Malauradament, ara per ara NLTK no disposa de corpus *chunkejats* per al català, de manera que farem les activitats següents per a l'anglès.

L'objectiu serà crear un *chunker* capaç de detectar els sintagmes nominals (NP).

El primer que farem, per a establir un valor mínim o *baseline* és crear un *chunker* que no *chunkeja* res i l'avaluarem (`programa-6-6.py`):

```
import nltk
from nltk.corpus import conll2000
cp=nltk.RegexpParser("")
test_sents=conll2000.chunked_sents("train.txt",chunk_types=['NP'])
print cp.evaluate(test_sents)
```

Aquest programa ens oferirà la sortida que teniu a continuació (la donada per la funció `evaluate`):

```
ChunkParse score:
    IOB Accuracy:  44.1%
    Precision:      0.0%
    Recall:         0.0%
    F-Measure:      0.0%
```

Abans d'analitzar aquests valors hem de comprendre els conceptes de *accuracy*, *precision*), *recall* i *F-measure*. En català no hi ha una distinció clara entre *accuracy* i *precision* i s'acostuma a fer servir el terme *precisió* per a tots dos conceptes. El concepte de *recall* tampoc no té una traducció clara en català, i per aquest motiu mantindrem els termes en anglès per a no donar lloc a confusions.

L'*accuracy* i *precision* mesuren el percentatges d'elements classificats correctament i en aquest sentit són mesures molt similars. Hi ha però, una diferència. El nostre sistema pretenia classificar correctament *chunks* de tipus NP i no n'ha classificat cap, tot i que com que molts dels elements del corpus de test estaven fora de *chunks* NP l'*accuracy* és del 44%, però la *precision* del 0%, ja que no ha identificat cap NP.

Aquest dos conceptes es poden entendre millor classificant les sortides del sistema en 4 tipus:

1) True Positive (TP): elements rellevants recuperats (en el nostre cas el nombre de NP classificats correctament).

2) False Positive (FP): elements no rellevants recuperats (en el nostre cas el nombre d'elements fora de NP classificats correctament).

3) False Negative (FN): elements rellevants no recuperats (en el nostre cas el nombre de NP no classificats correctament).

4) True Negative (TN): elements irrellevants no recuperats (en el nostre cas el nombre d'elements fora de NP no classificats correctament).

Així, tenim que l'*accuracy* es defineix com:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

i en canvi la *precision* es defineix com:

$$Precision = \frac{TP}{TP + TN}$$

El *recall* indica la quantitat d'elements classificats correctament respecte a la quantitat d'elements per classificar (en en nostre cas el nombre de NP classificats correctament respecte al nombre total de NP). Es defineix com:

$$Recall = \frac{TP}{TP + FN}$$

Finalment, la *F-measure* és una combinació de *precision* i *recall* per a poder tenir un únic indicador. Es defineix com la mitjana harmònica de *precision* i *recall*:

$$F\text{ - }measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

En alguns casos la mesura de *precision* i *recall* és equivalent (de fet, quan TN, FP i TN són zero a la força). Això es dóna en els casos que tots els elements han de ser classificats pel sistema. En el cas que ens ocupa, només volem classificar NP i no la resta. Si féssim que el nostre *chunker* detectés tots els tipus de *chunks* possibles tots dos valors coincidirien.

Amb les avaluacions dels *chunkers* següents podrem aprofundir més en aquestes mesures i acabar-les d'entendre.

Ara modificarem el programa anterior posant un senzill *chunker* que busqui etiquetes que comencin per lletres que són característiques dels sintagmes nominals (`programa-6-7.py`).

```
import nltk
from nltk.corpus import conll2000
gramatica=r"NP: {<[CDJNP].*>+}"
cp=nltk.RegexpParser(gramatica)
test_sents=conll2000.chunked_sents("train.txt",chunk_types=['NP'])
print cp.evaluate(test_sents)
```

La sortida d'aquest programa és la següent:

```
ChunkParse score:
    IOB Accuracy:  87.4%
    Precision:     69.7%
    Recall:        67.5%
    F-Measure:     68.6%
```

Ara tenim ja valors de *precision* i *recall*, ja que el programa és capaç de detectar NP; tot i això, fixeu-vos que *accuracy* és més alta, ja que aquesta mesura també té en compte tot allò que està fora de NP i que es classifica correctament.

Per ara encara no hem entrenat cap *chunker* a partir del corpus. El primer que farem serà entrenar un *chunker* a partir d'un etiquetador per unigrames (`programa-6-8.py`):

```
import nltk
from nltk.corpus import conll2000

class UnigramChunker(nltk.ChunkParserI):
    def __init__(self, train_sents):
        train_data = [[(t,c) for w, t, c in nltk.chunk.tree2conlltags(sent)]
                      for sent in train_sents]
        self.tagger = nltk.UnigramTagger(train_data)

    def parse(self, sentence):
        pos_tags = [pos for (word,pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
        conlltags = [(word, pos, chunktag) for ((word, pos), chunktag)
                     in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)
```

```
test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
unigram_chunker = UnigramChunker(train_sents)
print unigram_chunker.evaluate(test_sents)

ChunkParse score:
    IOB Accuracy:  92.9%
    Precision:     79.9%
    Recall:        86.8%
    F-Measure:     83.2%
```

Fixem-nos ara en el codi del programa, la classe `UnigramChunker` defineix dos mètodes: un constructor que es crida quan creem un `UnigramChunker` nou; i un mètode `parse`, que fem servir per a *chunkejar* noves oracions.

El constructor:

```
    def __init__(self, train_sents):
        train_data = [[(t,c) for w, t, c in nltk.chunk.tree2conlltags(sent)]
                      for sent in train_sents]
        self.tagger = nltk.UnigramTagger(train_data)
```

espera rebre una llista d'oracions d'entrenament en format d'arbre. El primer que fa és convertir cada arbre en una llista de triplets "paraula, etiqueta, chunk" fent servir la funció `tree2conlltags`. A partir de les dades convertides en aquest format entrena un etiquetador per unigrames. Aquest etiquetador s'emmagatzema en `self.tagger`, per a poder-lo fer servir més endavant.

El mètode `parse`:

```
    def parse(self, sentence):
        pos_tags = [pos for (word,pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
        conlltags = [(word, pos, chunktag) for ((word, pos), chunktag)
                     in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)
```

pren com a entrada una frase etiquetada. El primer que fa és treure les etiquetes de l'oració i etiquetar-les amb etiquetes de *chunk* del tipus IOB, fent servir el `self.tagger` entrenat i emmagatzemat en el constructor. Després combina aquestes etiquetes IOB amb l'oració etiquetada original per a obtenir una oració *chunkejada* amb etiquetes IOB. Finalment, amb la funció `conlltags2tree` la converteix en un format d'arbre.

Ara podem crear un *chunker* per bigrames, canviant el nom de la classe i fent servir un `BigramTagger` en comptes d'un `UnigramTagger`. Podem veure aquestes modificacions en el `programa-6-9.py`:

```
import nltk
from nltk.corpus import conll2000

class BigramChunker(nltk.ChunkParserI):
    def __init__(self, train_sents):
        train_data = [[(t,c) for w, t, c in nltk.chunk.tree2conlltags(sent)]
                      for sent in train_sents]
        self.tagger = nltk.BigramTagger(train_data)

    def parse(self, sentence):
        pos_tags = [pos for (word,pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
        conlltags = [(word, pos, chunktag) for ((word, pos), chunktag)
                     in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)

test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
bigram_chunker = BigramChunker(train_sents)
print bigram_chunker.evaluate(test_sents)
```

Si executem el programa observarem que assoleix un resultats lleugerament millors:

```
ChunkParse score:
    IOB Accuracy:  93.3%
    Precision:     82.3%
    Recall:        86.8%
    F-Measure:     84.5%
```

6. Aplicacions dels analitzadors fragmentals i dels analitzadors superficials

No totes les tasques del processament del llenguatge natural necessiten una anàlisi sintàctica completa. Una anàlisi no tan exhaustiva, ja sigui a escala d'anàlisi fragmental (*chunking*) o d'anàlisi superficial (*shallow parsing*) pot ser suficient per a nombroses tasques.

Així, l'anàlisi fragmental o *chunking* es fa servir amb èxit en les tasques següents:

- El *chunking* es fa servir sovint com a pas previ per a l'anàlisi superficial.
- En molts sistemes de reconeixement d'entitats amb nom (*named entity recognition*) el *chunking* és un dels passos inicials (Zhou i Su, 2002).

I l'anàlisi superficial o *shallow parsing* es fa servir en les tasques següents, que inclouen tant processament de llenguatge escrit com de parla:

- En alguns sistemes de traducció automàtica per transferència, és superficial aquesta transferència, és a dir, no es porta a terme una anàlisi sintàctica completa. Un exemple d'aquesta metodologia és el sistema Apertium (Armantano-Oller i altres, 2007).
- També s'han fet servir per a augmentar la robustesa d'un sistema de traducció automàtica de veu, dins del projecte Verbmobil (Wahlster, 2000).
- L'anàlisi superficial es fa servir per a reduir l'espai de cerca del analitzadors sintàctics profunds (Collins, 1996).
- Es fan servir per a sistemes de pregunta-resposta sobre Internet, en què és necessari processar gran quantitat de documents (Buchholz i Daelemans, 2001), (Srihari i Li, 1999).
- També es fa servir l'anàlisi superficial en mineria de textos (Sekimizu i altres, 1998).

Resum

En aquest mòdul hem après què és un *chunker* i com podem crear *chunkers* amb NLTK. L'anàlisi fragmental (*chunking*) i l'anàlisi superficial (*shallow parsing*) són dues tasques molt relacionades, tant que sovint es confonen. Tot i que aquest tipus d'anàlisi no és complet hi ha tot un seguit d'aplicacions que fan servir anàlisi fragmental o superficial amb èxit. En el mòdul següent veurem com podem portar a terme una anàlisi sintàctica més profunda.

Bibliografia

Abney, S. (1996). «Partial Parsing via Finite-State Cascades». A: «Proceedings of the ESSLLI'96 Robust Parsing Workshop».

Armantano-Oller, C.; Pérez-Ortiz, J.; Ramírez-Sánchez, G. i Sánchez-Martínez, F. (2007). «Apertium, una plataforma de código abierto para el desarrollo de sistemas de traducción automática». A: «Proceedings of the FLOSS International Conference 2007», (pàgs. 5–20). Servicio de Publicaciones de la Universidad de Cadiz. ISBN 978-84-9828-124-8. URL: `http://www.dlsi.ua.es/ mlf/docum/armentano07p.pdf`.

Buchholz, S. i Daelemans, W. (2001). «Complex Answers: A Case Study using a WWW Question Answering System». A: *Natural Language Engineering*.

Collins, M. (1996). «A new statistical parser based on bigram dependencies». A: «Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics».

Hammerton, J.; Osborne, M.; Armstrong, S. i Daelemans, W. (2002). «Introduction to Special Issue on Machine Learning Approaches to Shallow Parsing». A: *Journal of Machine Learning Research*, (2): pàgs. 551–558. URL: `http://jmlr.csail.mit.edu/papers/volume2/hammerton02a/hammerton02a.pdf`.

Sekimizu, T.; Park, H. i Tsujii, J. (1998). «Identifying the interaction between genes and gene products based on frequently seen verbs in medline abstracts».

Srihari, R. i Li, W. (1999). «Information extraction supported question answering». A: «Proceedings of TREC 8».

Tjong Kim Sang, E. i Bucholz, S. (2000). «Introduction to the CoNLL-2000 Shared Task: Chunking». A: «Proceedings of CoNLL-2000 and LLL-2000», (pàgs. 127–132). Lisboa. URL: `http://www.aclweb.org/anthology/W/W00/W00-0726.pdf`.

Wahlster, W. (ed.) (2000). *Verbmobil: Foundations of Speech-to-Speech Translation*. Springer.

Zhou, G. i Su, J. (2002). «Named Entity Recognition using an HMM-based Chunk Tagger». A: «Proceedings of the 4oth Annual Meeting of the Association for Computational Linguistics (ACL)», (pàgs. 473–480).