CREACIÓ D'UN *SOFTWARE* PRONOMINALITZADOR

Joan Gomà Cortés
Tutor: Damián Morales
La Salle Tarragona
2n Batxillerat
Curs 2021-2022

Resum

Partint del de l'interès per la programació i la llengua catalana, aquest treball de recerca consisteix en la creació d'un *software* que pronominalitza mitjançant pronoms febles els complements verbals d'oracions simples en català. Per elaborar-lo s'ha fet ús del llenguatge de programació Python per simplificar la relació amb l'eina que processa les oracions, anomenada Spacy. Un cop finalitzat el *software*, publicat en un repositori de GitHub, s'ha creat un lloc web i una API per visibilitzar el treball de recerca i rebre'n retroacció. En el treball es descriuen de forma detallada totes les fases de la creació del *software*, així com les dificultats i els reptes de futur.

Paraules clau: programació, pronoms febles, llengua catalana, Python, Spacy, web

Abstract

Based on the interest in programming and the Catalan language, this research work consists of the creation of a *software* that pronominalizes the verbal complements of simple sentences with Catalan weak pronouns. Python has been used to simplify the relationship with the sentence processing tool, called Spacy. Once the software is complete, which is published in a GitHub repository, a website and an API were carried out to make the research visible and get feedback. The project describes in detail all the phases of software creation, as well as the difficulties and challenges for the future.

Keywords: programming, weak pronouns, Catalan language, Python, Spacy, web

ÍNDEX

0. INTRODUCCIÓ

0.1 MOTIVACIONS

Els motius que m'han portat a escollir aquest treball han sigut molt diversos i seran exposats a continuació. Per començar, jo ja tenia una gran passió per la programació i la informàtica, sense la qual hauria sigut impossible plantejar-me aquest tema. Tot i que ja tenia una bona base del llenguatge de programació Python, adquirida gràcies a l'extraescolar que feia per mitjà de la plataforma Codelearn[1] i als concursos de programació Hp Codewars i Olimpíada Informàtica Catalana,[2] tenia una gran curiositat per veure com podia extrapolar tots aquests coneixements a l'àmbit de la lingüística computacional, que era totalment desconegut per a mi fins a aquell moment.

S'ha de dir que en un primer moment pot semblar que la lingüística i la informàtica no vagin molt lligades, però gràcies a unes converses que vaig tenir amb el meu professor de Llengua castellana, Damián Morales, sobre la recerca que ell estava realitzant en el seu treball de tesi, se'm va obrir un món totalment nou.

Al cercar informació sobre la lingüística computacional en català, vaig veure que, malauradament, hi havia projectes i materials, però no tants com jo em pensava. Llavors ja vaig tenir clar que el meu treball de recerca s'havia de centrar en aquest àmbit. A continuació, se'm va acudir la idea de fer un pronominalitzador de pronoms febles automàtic i posar-lo al servei de la comunitat, tant per a l'usuari com per al programador que busca recursos en aquest àmbit.

Com que aquesta idea en un primer moment pot semblar una mica fora de lloc, vaig demanar consells als investigadors del Barcelona Supercomputing Center, on estava fent el curs Bojos per la supercomputació de la Fundació Catalunya la Pedrera.[3] Em van concedir una reunió amb la investigadora Marta Villegas i el seu equip, i gràcies al seu suport i els mitjans que em van

[1] Vegeu: https://codelearn.es/.
[2] Vegeu: https://hpcodewarsbcn.com/ i https://olimpiada-informatica.cat/.
[3] Vegeu: https://www.fundaciocatalunya-lapedrera.com/ca/bojos-ciencia/cursos/supercomputacio.

presentar per realitzar la meva investigació, em vaig veure en cor de començar el treball ben encaminat.

Una altra qüestió que em va motivar a dur a terme aquesta idea va ser que tenia curiositat per veure com aconseguiria millorar la meva faceta lingüística per mitjà d'un àmbit de coneixement que m'apassionava.

0.2 OBJECTIUS

Aquest treball de recerca té els objectius següents:

1. Aconseguir programar un *software* capaç de detectar complements verbals i pronominalitzar-los individualment i de forma binària. Assolir que aquest *software* funcioni correctament en un gran percentatge de casos, tenint en compte les limitacions que tenen els ordinadors.
2. Conèixer i aprendre com funciona el mòdul Spacy, que és la base del meu projecte, creat pels investigadors del Barcelona Supercomputing Center que treballen en l'àmbit de *text mining*.[4]
3. Dur a terme la recerca d'una manera transparent i accessible, per tal de poder-ne fer difusió i que qualsevol persona pugui aprofitar el meu treball i millorar-lo.
4. Crear una pàgina web perquè qualsevol persona pugui fer ús del *software* i aclarir dubtes sobre com es pronominalitza en català.

[4] Vegeu: https://www.bsc.es/ca/discover-bsc/organisation/scientific-structure/text-mining.

MARC TEÒRIC

1.TEORIA SOBRE ELS PRONOMS

1.1 Què són els pronoms febles?

Segons el Diccionari de l'Institut d'Estudis Catalans, un pronom és una "categoria lèxica constituïda per mots generalment variables, que poden tenir caràcter substantiu, adjectiu o adverbial i que presenten un significat ocasional, delimitat a partir de relacions díctiques o anafòriques", mentre que un pronom feble és "un pronom inaccentuat que va immediatament al davant o al darrere del verb" (DIEC, 2021).

L'objectiu del meu treball de recerca és crear un *software* capaç d'identificar complements del verb i pronominalitzar-los correctament, incloent-hi combinacions binàries. Per tant, en aquest apartat em centraré exclusivament a explicar detalladament cada pronominalització dels complements i les seves combinacions, ja tenint en compte els problemes que poden sorgir a l'hora de dur a terme la part pràctica. Explicaré els següents complements i les seves respectives pronominalitzacions: complement directe, complement indirecte, complement de règim verbal, complement predicatiu, atribut i complement circumstancial.

Per elaborar aquests continguts, em basaré en els apunts de primer de Batxillerat facilitats per la meva professora de Llengua i Literatura catalana Marina Torné Izquierdo, i en la Gramàtica Catalana de l'Institut d'Estudis Catalans (2018).

1.2 Complement directe

Segons l'Optimot (2021), el complement directe (CD) és aquell que expressa sobre què o qui recau directament l'acció del verb. A continuació, hi ha una llista amb totes les possibilitats amb les quals el complement directe pot estar introduït, i la seva pronominalització corresponent. En el cas del complement directe, existeixen tres possibilitats a l'hora de ser pronominalitzat.

1. Complement directe determinat: Sempre anirà introduït per un article determinat o un determinant demostratiu. Els possibles pronoms febles que substituiran aquest complement són: *el*, *la*, *els*, *les*.

Exemples:

- El nen compra **les patates** => El nen **les** compra
- La Maria compra **aquella pilota** => La Maria **la** compra

2. Complement directe indeterminat: Sempre anirà introduït per un determinant quantitatiu, un determinant indefinit, un numeral o res. Sempre es pronominalitzarà amb el pronom *en*. En aquest cas, sempre que el complement estigui introduït, haurem d'incloure el determinant o el numeral després del verb.

   Exemples:

   - El nen compra **moltes patates** => El nen **en** compra **moltes**
   - El pare compra **menjar** => El pare **en** compra
   - La mare compra **tres pastanagues** => La mare **en** compra tres

3. Complement directe neutre: Sempre anirà introduït per *això*, *allò* o una oració subordinada substantiva. En aquest últim cas, el pronom corresponent per pronominalitzar el complement directe neutre serà el pronom *ho*.

   Exemples:

   - Després del migdia, farem **allò** => Després del migdia, **ho** farem
   - Vull **que vinguis** => **Ho** vull

   Tot i que s'explicarà amb deteniment a l'apartat corresponent de la part pràctica, identificar els complements directes que siguin una oració subordinada portarà força problemes, i degut a l'abast d'aquest projecte, aquests complements no es detectaran.

1.3 Complement indirecte

Segons l'Optimot (2021), el complement indirecte (CI) és el complement verbal que expressa la persona o l'objecte que rep la conseqüència de la significació del verb. Aquest complement només té una possibilitat a l'hora de ser pronominalitzat.

El complement indirecte sempre estarà introduït per la preposició *a* o *per a*. Depenent de si a la persona o objecte que el complement s'està referint és singular o plural, estarà pronominalitzat pel pronom *li* o *els*.

Exemples:

- Porta això **al teu germà** => Porta-**li** això
- Escriu cartes **als teus pares** => **Els** escriu cartes

1.4 Complement de règim verbal

Segons l'Optimot (2021), el complement de règim és aquell complement que va introduït per una preposició i és seleccionat pel verb.

Aquest complement només es pot pronominalitzar de dues maneres. Si està introduït per la preposició *de*, es pronominalitzarà amb el pronom *en*. En canvi, si està introduït per qualsevol altre preposició, es pronominalitzarà amb el pronom *hi*.

Exemples:

- M'han parlat molt **de** tu => Me **n**'han parlat molt
- M'he acostumat **a** fer els deures a la tarda => M'**hi** he acostumat

1.5 Complement predicatiu

Segons l'Optimot (2021), el complement predicatiu expressa una propietat del subjecte o del complement directe (sempre que l'oració no sigui copulativa, perquè, en aquest cas, parlaríem d'atribut). Generalment, els complements predicatius són sintagmes adjectivals, nominals i preposicionals i, si escau, concorden en gènere i nombre amb el subjecte o el complement directe.

Aquest complement generalment estarà pronominalitzat pel pronom *hi*. Hi ha certes excepcions que cal tenir en compte per pronominalitzar-lo correctament. En el cas que es vulgui emfatitzar el complement predicatiu, s'ha d'utilitzar el pronom *en*, tal i com es veu en el primer exemple. Aquest complement també s'haurà de pronominalitzar pel pronom *en*, quan estigui introduït per un dels següents verbs: *fer-se, dir-se, elegir, nomenar*.

Exemples:

- El gos **en** va molt, de **brut**
- Els nens juguen **contents** => Els nens **hi** juguen
- L'han nomenada **alcaldessa de la seva ciutat** => La **n**'han nomenada

1.6 Complement circumstancial

Segons l'Optimot (2021), el complement circumstancial és un complement verbal que, respecte a la significació del verb, expressa circumstàncies de mode, de lloc, de temps, etc.

A l'hora de pronominalitzar aquest complement, només existeixen dues possibilitats. Generalment, estarà pronominalitzat amb el pronom *hi*, excepte quan el complement estigui introduït amb la preposició *de* i el complement circumstancial sigui de lloc, que es pronominalitzarà amb el pronom *en*.

Exemples:

- Esmorzarem **a les set** => **Hi** esmorzarem
- Treu la carn **de la nevera** => Treu-**ne** la carn

Respecte a la part pràctica, aquí cal fer dues petites observacions que poden ser una font de problemes. Primer de tot, no només en tindrem prou a detectar un complement circumstancial, sinó que també s'haurà de detectar si és de lloc. Això, tot i que pot semblar una minúcia, pot ser molt complicat, ja que pot comportar entrenar un model únicament i exclusivament per fer aquesta tasca. L'altre problema és que certes oracions són morfològicament idèntiques, però unes són complement indirecte, i les altres complement circumstancial de lloc, tal i com es pot veure en l'exemple.

Exemples:

- Escriu una carta **a la nena** (CI) ≠ Escriu una carta **a la biblioteca** (CCL)

És per això que aquestes excepcions, que són molt òbvies per una persona, poden ser molt complicades a l'hora de fer que funcionin correctament programant.

1.7 Atribut

Segons el DIEC, l'atribut és un complement format per un mot o grup de mots que qualifiquen o determinen un subjecte en les oracions amb verbs com ara *ésser*, *estar* o d'altres de semblants.

Aquest complement es pot pronominalitzar de tres maneres diferents.

1. Atribut determinat: Sempre que estigui introduït per un article determinat o un determinant demostratiu. Els possibles pronoms febles que substituiran aquest complement són: *el*, *la*, *els*, *les*.

   Exemple: Aquests nois són **els meus amics** => Aquests nois **els** són

2. Atribut emfàtic: Tal i com passa amb el complement predicatiu, si es vol emfatitzar el complement, s'utilitzarà el pronom *en*.

   Exemple: Que **n**'estava de content, el dia del seu aniversari.

3. Els altres casos: La resta de casos, el complement es pronominalitzarà amb el pronom *ho*.

   Exemple: Aquell nen és **molt interessant** => Aquell nen **ho** és

1.8 Combinacions binàries

Les combinacions binàries son aquelles combinacions de dos pronoms febles junts que s'utilitzen quan es volen substituir dos complements alhora. Fer un ús correcte d'aquestes combinacions pot ser una mica confús, fins i tot per a la gent catalanoparlant, ja que hi ha certes excepcions que cal saber per substituir els complements respectius d'una manera correcta.

Com que la implementació que es fa a la part pràctica no segueix el procés que faria una persona, no s'explicarà tota la teoria de pronominalitzar, ja que després no s'utilitza per a res, a diferència de tots els altres complements. Per fer-se una idea de totes les possibilitats que hi ha, vegeu la taula següent.

Primer pronom		Segon pronom												
		hi	**en**	**ho**	**les**	**la**	**els (CD)**	**el**	**els (CI)**	**li**	**ens**	**em**	**us**	**et**
es		s'hi	se'n se n'	s'ho	se les	se la se l' / se la	se'ls	se'l se l'	se'ls	se li	se'ns	se'm se m'	se us	se't se t'
		-s'hi	-se'n	-s'ho	-se-les	-se-la	-se'ls	-se'l	-se'ls	-se-li	-se'ns	-se'm	-se-us	-se't
et		t'hi	te'n te n'	t'ho	te les	te la te l' / te la	te'ls	te'l te l'	te'ls	te li	te'ns	te'm te m'		
		-t'hi	-te'n	-t'ho	-te-les	-te-la	-te'ls	-te'l	-te'ls	-te-li	-te'ns	-te'm		
us		us hi	us en us n'	us ho	us les	us la us l' / us la	us els	us el us l'	us els	us li	us ens	us em us m'		
		-vos-hi -us-hi	-vos-en -us-en	-vos-ho -us-ho	-vos-les -us-les	-vos-la -us-la	-vos-els -us-els	-vos-el -us-el	-vos-els -us-els	-vos-li -us-li	-vos-ens -us-ens	-vos-em -us-em		
em		m'hi	me'n me n'	m'ho	me les	me la me l' / me la	me'ls	me'l me l'	me'ls	me li				
		-m'hi	-me'n	-m'ho	-me-les	-me-la	-me'ls	-me'l	-me'ls	-me-li				
ens		ens hi	ens en ens n'	ens ho	ens les	ens la ens l' / ens la	ens els	ens el ens l'	ens els	ens li				
		-nos-hi 'ns-hi	-nos-en 'ns-en	-nos-ho 'ns-ho	-nos-les 'ns-les	-nos-la 'ns-la	-nos-els 'ns-els	-nos-el 'ns-els	-nos-els 'ns-els	-nos-li 'ns-li				
li	*general*	li hi	li'n li n'	li ho	les hi	la hi	els hi	l'hi						
		-li-hi	-li'n	-li-ho	-les-hi	-la-hi	-los-hi 'ls-hi	-l'hi						
li	*valencià*	li hi	li'n li n'	li ho	li les	li la li l' / li la	li'ls	li'l li l'						
		-li-hi	-li'n	-li-ho	-li-les	-li-la	-li'ls	-li'l						
els (CI)		els hi	els en els n'	els ho	els les	els la els l' / els la	els els	els el els l'						
		-los-hi 'ls-hi	-los-en 'ls-es	-los-ho 'ls-ho	-los-les 'ls-les	-los-la 'ls-la	-los-els 'ls-els	-los-el 'ls-el						
el		l'hi	le'n el n'											
		-l'hi	-l'en											
els (CD)		els hi	els en els n'											
		-los-hi 'ls-hi	-los-en 'ls-en											
la		la hi	la'n la n'											
		-la-hi	-la'n											
les		les hi	les-en les n'											
		-les-hi	-les-en											
en		n'hi												
		-n'hi												

2. PROGRAMACIÓ

Aquest treball de recerca ha sigut desenvolupat utilitzant el llenguatge de programació Python. Tots el que s'explica són coneixements adquirits prèviament gràcies a la plataforma Codelearn.[5]

2.1 Introducció a Python

2.1.1 Què és i per què Python?

Python és un llenguatge de programació interpretat, que ha anat guanyant popularitat en els últims anys. Aquest fet ha succeït gràcies a la fàcil sintaxi que té a l'hora de programar, i com a conseqüència d'això, aquest llenguatge ha sigut un dels primers llenguatges apresos per a molts programadors.

Com que cada vegada més gent aprèn Python per iniciar-se en el món de la programació, surten més mòduls i llibreries específiques, termes que s'explicaran més endavant, que permeten fer tasques concretes amb aquest llenguatge. Gràcies a això, qui sap programar amb Python no cal que aprengui gaires més llenguatges de programació per fer tot tipus de tasques.

Per exemple, en el món de desenvolupament de pàgines web, el llenguatge utilitzat per excel·lència era i és Javascript. Tot i ser molt popular i útil, ha anat perdent protagonisme per culpa de Python, ja que s'ha desenvolupat un *framework,* anomenat Flask,[6] que només amb coneixements de Python, HTML i CSS pot desenvolupar una pàgina web completa.

És per aquestes raons descrites anteriorment, i sobretot per l'abundància de llibreries i continguts, que he decidit dur a terme la meva recerca utilitzant Python.

2.1.2 Aspectes bàsics de programació en Python

A continuació s'explicaran tots els aspectes bàsics de programació que s'han de tenir interioritzats per poder entendre la part pràctica d'aquest treball d'una manera completa.

[5] Vegeu: https://codelearn.cat/.
[6] Vegeu: https://flask.palletsprojects.com/en/2.0.x/.

2.1.3 Tipus de dades i variables

Quan es programa, s'utilitzen tot tipus de dades, i mitjançant operacions s'intenta obtenir el resultat esperat.

Els tipus de dades primitives que qualsevol programador ha de conèixer són: nombres enters, nombres decimals, valors booleans i els caràcters, que s'anomenen amb l'abreviatura (int), (float), (bool) i (char), respectivament.

Les variables són elements que ens permeten guardar dades com les que s'acaben d'anomenar i d'altres més complexes. El procés de declarar variables amb Python és força senzill comparat amb altres llenguatges de programació. Una variable es declararia de la següent manera:

NOM VARIABLE = VALOR

Exemple:

```
temperatura = 15.7
```

2.1.4 Funció print

La funció *print* mostra per pantalla els valors que se li han passat per paràmetres, concepte que explicaré més endavant. El missatge que surt per pantalla pot ser qualsevol variable, valor o expressió. Utilitzant la coma entre cada valor, podem mostrar-ne molts.

Tot i que aquesta instrucció és molt senzilla, és extremadament útil i necessària per a qualsevol programador, ja que per desenvolupar qualsevol programa es necessita veure si *l'output* és el correcte, i en cas de no ser-ho, la instrucció *print* ens ajuda a trobar on hi ha l'error.

Exemple:

```
habTarragona = 132299
print ("Tarragona té", habTarragona)
```

>> Output: Tarragona té 132299

2.1.5 Operadors en Python

Per poder manipular les variables, existeixen certs operadors que ens permeten fer les operacions necessàries.

Els operadors bàsics per operar amb nombres son els següents:

- **Suma: +**
- **Resta: -**
- **Multiplicació: ***
- **Divisió: /**
- **Divisió entera: //**
- **Residu: %**
- **Potència ****

Exemple:

```
temperatura = 22
temperatura = temperatura + 2
print (temperatura)
```

>> Output: 24

Cal puntualitzar que aquests operadors no són exclusius per operar amb nombres, ja que, per concatenar paraules o fer tasques més concretes, els operadors anomenats anteriorment poden funcionar igualment.

Perquè el codi es pugui executar i no tingui errors, cal que els dos valors amb els quals estem operant siguin del mateix tipus. Això vol dir que, si tenim una variable de tipus *string*, que s'explicarà més endavant, que està representant un nombre (s = "123"), i volem sumar al valor de la variable **s** el nombre enter 10, no ho podrem fer com ho faríem sumant dos nombres. Per poder dur a terme la tasca descrita anteriorment, hauríem de fer un *casting*. Aquest procés consisteix a transformar el tipus de qualsevol variable primitiva a un altre. Per poder fer això, escriurem el nou tipus primitiu al que volem que la variable es transformi, i entre parèntesis, la variable que volem transformar.

NOVA VARIABLE = NOU TIPUS(VARIABLE ANTIGA)

Exemple:

```
s = "123"
n = 10
resultat = int(s) + 10
print (resultat)
```

>> Output: 133

2.1.6 Funcions en Python

Una funció és una seqüencia d'instruccions que fan una tasca concreta i que identifiquem amb un nom. Cada vegada que escrivim el nom de la funció al nostre programa, s'executaran totes les instruccions escrites a dins de la funció.

Tot i que això és força útil, el seu ús estaria força limitat, ja que faria exactament el mateix sempre. És per això que existeixen les funcions amb paràmetres. Aquestes funcions les programem de manera que interpretem els paràmetres com una variable, i després posem el nom de la variable al nostre codi en comptes de posar un valor qualsevol.

Si volem que una funció sumi dos valors, definirem la funció amb els dos paràmetres corresponents i a dins farem els càlculs corresponents. Després podrem retornar el valor o ensenyar-lo per pantalla.

L'estructura que segueixen les funcions és la següent:

Sense paràmetres:

```
def nom_funció():
    instruccions
```

Amb paràmetres:

```
def nom_funció(parametres):
    instruccions
```

Exemple:

```
def sumar(n, m):
    suma = int(n) + int(m)
    print(suma)

sumar(2, 3)
```

>> Output: 5

2.1.6.1 Instrucció *return*

Aquesta instrucció serveix per retornar algun valor i finalitzar la funció. Tot i que aquesta instrucció és força senzilla, s'ha de ser conscient que, quan la

utilitzem, sortim automàticament de la funció, per molt que tinguem més codi escrit sota. Aquest valor que retornem majoritàriament serà emmagatzemat en una variable.

Tot i que la instrucció *return* és molt senzilla i el seu funcionament es veu molt ràpid, es poden arribar a fer algoritmes molt complexos mitjançant la recursivitat.[7]

Exemple:

```
def sumar(n, m):
    suma = int(n) + int(m)
    return suma

suma = sumar(2, 3)
print(suma + sumar(4, 55))
```

>>Output: 64

2.1.7 Expressions Booleanes

Una expressió booleana és una expressió que produeix un resultat booleà (cert o fals). En programació, s'utilitza un operador de comparació, i aquest compara dos valors i retorna cert si aquella expressió és certa, o falsa si no ho és.

2.1.7.1 Operadors de comparació

Per poder avaluar tota mena d'expressions, existeixen els operadors de comparació, que són molt utilitzats i que cal conèixer per poder realitzar quasi qualsevol tasca. Aquests són els següents:

- Més gran: **>**
- Més petit: **<**
- Més gran o igual: **>=**
- Més petit o igual: **<=**
- Igual: **==**
- Diferent: **!=**

[7] Vegeu-ne algun exemple a: https://www.geeksforgeeks.org/recursion/.

Cal remarcar que tots aquests operadors s'utilitzen exclusivament per comparar elements. No es poden utilitzar per declarar variables ni per cap altra mena de tasca.

2.1.7.2 Operadors lògics

Per poder concatenar diverses expressions booleanes, existeixen els operadors lògics. Aquests son els següents:

and: avaluarà l'expressió com a certa quan tant A com B siguin certes. En cas contrari, l'avaluarà com a fals. Per utilitzar-lo programant s'escriu **and**.

or: avaluarà l'expressió com a certa quan A o B, o els dos alhora, siguin expressions certes. En cas contrari, l'avaluarà com a fals. Per utilitzar-lo programant s'escriu **or**.

not: avaluarà l'expressió com a certa quan l'expressió A sigui falsa. En cas contrari, l'avaluarà com a fals. Per utilitzar-lo programant s'utilitza el símbol **!**.

xor: avaluarà l'expressió com a certa quan només una expressió entre A i B sigui certa. En cas contrari, l'avaluarà com a fals. Per utilitzar-lo programant s'utilitza el símbol **^**.

2.1.7.3 Estructures condicionals

Les estructures condicionals són expressions que, mitjançant una expressió booleana, determinen si s'ha de dur a terme una tasca o una altra.

Per poder programar utilitzant els condicionals, hi ha 3 expressions que permeten avaluar expressions booleanes. Són els següents:

IF: Serveix per comprovar si una expressió booleana és certa o falsa, i depenent del resultat, fer una tasca o una altra.

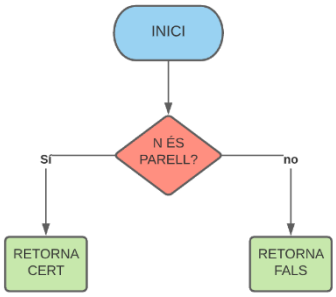
ELIF: És molt semblant a la instrucció **IF**, amb la diferència que només es pot utilitzar després que la instrucció **IF** hagi avaluat l'expressió corresponent com a falsa.

ELSE: Només es pot utilitzar com a última opció, executant-se només si els **IF** previs han avaluat totes les expressions com a falses.

L'estructura és la següent:

```
if condició:
    instruccions
else:
    instruccions
```

A continuació, vegeu un exemple on hi ha un diagrama de flux amb el codi corresponent d'una funció que determina si un nombre és parell o senar.




```
def parell(n):

    if n % 2 == 0:
        return "Parell"
    else:
        return "Senar"
```

Il·lustració 2. Diagrama funció parell

2.1.8 Estructures iteratives: While i For

Les estructures iteratives o bucles ens permeten repetir el mateix bloc de codi les vegades que calgui. Cada vegada que es comença un bucle s'anomena iteració, d'aquí el nom. Aquestes estructures són extremadament útils a l'hora de programar.

2.1.8.1 *While*

Una de les estructures iteratives que cal conèixer és ***while***. Aquest bucle s'executa mentre una condició sigui certa. És per això que quan es programa i es fa ús d'aquest bucle, s'ha de vigilar que aquesta condició, una vegada s'ha entrat en el bucle, en algun moment deixi de ser certa, ja que en un cas contrari, s'entraria en un bucle infinit. L'estructura és la següent:

```
while condicio:
    instruccions
```

Exemple:

```
i = 0
s = ""
while i < 10:
    s += str(i) + " " #afegim un nombre tipus string a la variable s
    i += 1            #incrementem la variable 1
print (s)
```

>>Output: 0 1 2 3 4 5 6 7 8 9

Si en aquest últim exemple no suméssim 1 a la variable i, el programa es col·lapsaria.

2.1.8.2 *For*

L'altra estructura iterativa per excel·lència és el bucle ***for***. Gràcies a la versatilitat que té a l'hora d'iterar per tipus de dades compostes, termes que s'explicaran més endavant, és un bucle que s'utilitza molt. El funcionament d'aquest bucle és força intuïtiu: itera per cada subelement que té un element més gran. Per exemple, podem iterar per cada lletra que té una paraula. També es pot fer un ***for*** que faci la mateixa feina que la instrucció ***while***, utilitzant la instrucció ***range()***. Com a paràmetres d'aquesta funció s'hi posaran dos nombres, i iterarem des del primer inclòs fins a l'últim sense incloure. En cas que només posem un nombre, s'iterarà des de 0 fins al nombre.

L'estructura és la següent:

```
for subelement in element:
    instruccions
```

Cal remarcar que *subelement* és una variable que canvia de valor a cada iteració.

Exemple 1:

```
s = "hola"
for lletra in s:
    print(lletra)
```

Output:

>>h
>>o
>>l
>>a

Exemple 2:

```
s = ""
for num in range(10):
    s += str(num) + " "
print(s)
```

>>Output: 0 1 2 3 4 5 6 7 8 9

2.1.9 Tipus de dades compostes

Tal i com s'ha vist, existeixen diferents tipus de dades primitives. Les dades compostes són força interessants, ja que estan formades per dades primitives.

2.1.9.1 Cadenes

Les cadenes, popularment conegudes com a *strings*, són un tipus de dada que s'utilitza molt i cal saber operar i manipular amb profunditat, sobretot en aquest treball, ja que quasi tota l'estona s'estaran manipulant variables que representaran cadenes.

Qualsevol subcadena d'una cadena s'anomena segment. Per tal de seleccionar subcadenes, ho farem accedint als índexs corresponents de la cadena. Per fer això, primer de tot necessitem tenir clar com funciona la indexació en programació. En comptes de començar a enumerar per l'1, es comença a enumerar pel 0. És per aquest fet que cal vigilar i no cometre errors. Per indexar una cadena es faria de la següent manera:

```
cadena = "prova"
subcadena = cadena[indx1 : indx2]
```

La subcadena que crearem estarà composta des de la posició inicial inclosa fins a la última sense incloure.

```
cadena = "SalleTarragona"
subcadena = cadena[0:5]
```

En el cas anterior, el valor de la variable subcadena seria "Salle".

Per accedir a un caràcter d'una cadena, s'haurà de saber la posició en què es troba el caràcter. Cal recordar sempre que es comença a enumerar per 0. La sintaxis per fer-ho és la següent:

```
s = "hola"
lletra = s[índex]
```

Per comprovar si una cadena és una subcadena d'una altra, s'utilitza l'operador ***in***. Per fer ús d'aquest operador, s'utilitzen els condicionals. És per això que el resultat de l'operació sempre serà un valor booleà. Aquest operador s'utilitza de la següent manera:

```
if subcadena in cadena:
    instruccions
else:
    instruccions
```

Python ja té creades per defecte unes funcions que son força útils per manipular cadenes. És cert que memoritzar-les totes és molt complicat, i és per això que qualsevol programador hauria de saber programar aquestes funcions per no dependre de res. Aquí he fet un recull d'unes quantes que són força útils:

- **Cadena.count("a"):** Retorna un enter que és la quantitat de "a" que hi ha a la cadena.
- **Cadena.capitalize():** Retorna la mateixa cadena amb la primera lletra en majúscula.
- **Cadena.endswith("a"):** Retorna True / False, depenent de si la cadena acaba amb la subcadena indicada.
- **Cadena.startswith("a"):** Retorna True / False, depenent de si la cadena comença amb la subcadena indicada.
- **Cadena.replace("a", "b"):** Retorna la cadena reemplaçant les "a" per "b".
- **Cadena.islower():** Retorna True si la cadena és tota en minúscules.
- **Cadena.isupper():** Retorna True si la cadena és tota en majúscules.

2.1.9.2 Llistes

Les llistes són elements que ens permeten guardar tot tipus de valors en una sola variable. Són molt similars a les cadenes, amb la diferència que les cadenes només poden emmagatzemar caràcters.

Crear una llista amb Python es pot fer de múltiples maneres, però una de les més senzilles és incloure els elements entre claudàtors i separats per comes, tal i com es veu a continuació.

```
llista = [1, 2, 3, 4]
```

Un fet que cal remarcar és que una llista pot tenir diferents tipus de dades guardats. Per exemple, pot emmagatzemar una llista, una cadena i un nombre enter, tal i com es veu a l'exemple següent.

```
llista = [[1, 2, 3, 4], "hola", 23]
```

La sintaxi per accedir als elements d'una llista és exactament igual que la que s'utilitza per accedir als caràcters de les cadenes. La sintaxi és la següent:

```
element = llista[índex]
```

En el cas que l'índex tingui un valor negatiu, es començarà a indexar pel darrere. Aquí, però l'última posició estarà representada pel nombre **-1.**

Per saber la llargada d'una llista, s'utilitza la funció ***len()***. Per fer ús d'aquesta funció, passarem per paràmetres la llista de la qual vulguem saber la llargada. Aquesta funció també es pot fer servir per les cadenes. La sintaxis és la següent:

```
llargada = len(llista)
```

2.2 Spacy

2.2.1 Què és Spacy?

Spacy és una llibreria de codi obert que serveix per extreure informació addicional d'un text. Aquesta informació pot formar part d'un ventall molt ampli, com podria ser de què va el text, o quines relacions hi ha entre les entitats d'aquest text. El camp que es dedica a investigar això mitjançant la

intel·ligència artificial es diu processament del llenguatge natural i es coneix per NLP per les seves sigles en anglès (Natural Language Processing).

Jo vaig descobrir aquesta llibreria gràcies a l'ajuda dels investigadors del Barcelona Supercomputing Center. Ells em van proporcionar un model que havien entrenat feia molt poc, ja disponible a la pàgina oficial d'Spacy. Per tant, per aprendre com funcionava, em vaig llegir la documentació de l'Spacy,[8] i em vaig posar a treballar.

2.2.2 Com s'implementa en un programa?

Primer de tot, el que s'ha de fer és instal·lar el programa Spacy com a tal, i després s'ha d'instal·lar una *trained pipline,* que serà l'element que determinarà l'idioma en què s'analitzarà el text. És a dir, l'Spacy ens ajudarà a gestionar la *pipeline* descarregada.

Quan es tenen els dos elements descarregats, implementar-los en el programa pertinent és força fàcil. Per fer-ho, escriurem la instrucció *import spacy*, i després, en una variable de nom **nlp,** hi guardarem el que ens retorni la funció *spacy.load(nom de la pipline).* Aquesta variable representarà un objecte que tindrà tota la informació necessària per analitzar textos. El nom de la variable pot ser el que cadascú vulgui, però al formar part d'un procés protocol·lari de la llibreria, es recomana que s'utilitzin els noms preestablerts en la documentació, ja que el codi queda més entenedor.

Una altra eina que ens proporciona Spacy és un visualitzador, de nom displacy, que ens permet veure l'anàlisi que fa la *pipline* d'una manera més amable. Per importar aquesta part de la llibreria, s'ha d'incloure la instrucció *from spacy import displacy*. Després de fer tot això, el nostre codi tindria el següent aspecte:

```
import spacy
from spacy import displacy

nlp = spacy.load("ca_base_web_trf")
```

[8] Vegeu: https://spacy.io/usage/spacy-101.

2.2.3 Com s'utilitza?

Per poder analitzar un text, el que s'ha de fer és cridar l'objecte nlp amb el text corresponent, i guardar el que retorna aquest objecte en una variable. Aquest procés es diu *"tokenitzar"* un text. És molt important saber que les paraules que formen aquesta variable ja no són de tipus *string*, i que per poder operar amb elles s'hauran de fer les conversions respectives.

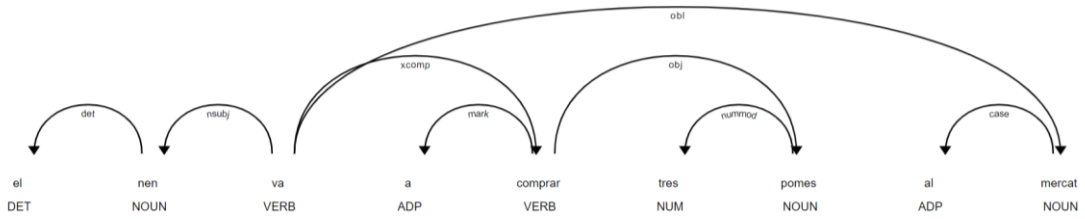
Per exemple, si es volgués analitzar la frase "el nen va a comprar tres pomes al mercat", ho faríem de la següent manera:

```
doc = nlp("el nen va a comprar tres pomes al mercat")
```

Tot i que ara ja es tindria el text analitzat, encara no es pot accedir a les dades que ens interessen.

En el cas que es volgués visualitzar el text amb el *displacy*, s'hauria de fer ús de la instrucció *displacy.serve(doc, style="dep").* Aquesta funció agafa el document modificat per l'objecte *nlp*, i després s'ha d'especificar per paràmetres l'element que es vol que ressalti. En el meu cas, m'interessa que marqui totes les dependències.

Quan s'executa el codi, displacy obre com una mena de servidor local, i si anem al navegador i copiem la url: http://localhost:5000/, es veuria el següent:



Il·lustració 3. Oració analitzada

S'ha de dir que el visualitzador que té l'Spacy és bastant clarificador, ja que d'una manera molt entenedora ajuda a veure com s'està analitzant el nostre text. També va molt bé veure el que ha generat, ja que quan es programa només hi ha variables i elements, i al principi veure-ho amb el displacy pot ajudar a aclarir dubtes.

Tot i això, encara no es pot accedir als elements i a la informació que contenen des del nostre codi. És per això que en el següent apartat s'explicarà com s'accedeix a les dependències, als complements representats, etc.

2.2.4 Com s'accedeix a la informació?

Abans d'explicar com s'accedeix als elements, m'agradaria explicar una mica què és el que ha generat l'objecte **nlp**. Per fer-se una idea, el nostre document ara està separat com en una llista per paraules. Aquestes paraules no són cadenes normals, tal i com he comentat abans, sinó que cada paraula és un objecte que té diferents atributs, com poden ser: categoria morfològica, dependències, etc. Aquests atributs són el que realment ens interessa, ja que donaran la informació necessària per poder pronominalitzar els complements de manera correcta.

Per poder accedir a la informació necessària, s'haurà d'iterar amb un bucle **for** pel nostre document modificat, i accedir als elements que es vulguin consultar de la manera que ens diu la documentació d'Spacy. Per exemple, si volem accedir a les dependències, s'haurà de fer ús de l'atribut **dep_**. Per veure tots els possibles atributs als quals es pot accedir visiteu: https://spacy.io/api/token#attributes.

Amb el següent exemple de codi clarifiquem els conceptes anteriors.

```
import spacy

nlp = spacy.load("ca_base_web_trf")
doc = nlp("el nen va a comprar tres pomes al mercat")

def frase():
    for token in doc:
        print(token, token.dep_, token.pos_, [child for child in token.children])
frase()
```

OUTPUT:

- el det DET []
- nen nsubj NOUN [el]
- va ROOT VERB [nen, comprar, mercat]
- a mark ADP []

- comprar xcomp VERB [a, pomes]
- tres nummod NUM []
- pomes obj NOUN [tres]
- al case ADP []
- mercat obl NOUN [al]

2.3 Git i GitHub

2.3.1 Què és Git i per què he decidit utilitzar-lo

Git[9] és el *software* de control de versions més utilitzat en tot el món. Git és un programari de codi obert i va ser desenvolupat per Linus Torvalds, el desenvolupador del Kernel de Linux.

Aquest gestor de projectes ens permet tenir un control absolut de totes les versions que hi ha hagut al llarg del temps en un projecte, de manera que pots accedir a versions específiques en qualsevol moment.

Aquesta eina permet fer el següent:

- Tenir un control del desenvolupament del *software*.
- Col·laborar en el mateix codi amb diverses persones d'una manera molt fàcil.

2.3.2 Què és GitHub?

GitHub[10] és una plataforma web on es poden allotjar els repositoris desenvolupats amb git. Els repositoris són els projectes que es desenvolupen utilitzant git. Tots aquests repositoris, una vegada estan allotjats a GitHub, són de domini públic, a no ser que es tingui un compte de pagament.

El gran potencial de GitHub recau en què tota la comunitat de programadors s'ajuda i comparteix codis, de manera que es pot aprendre de la feina dels altres i la gent pot aprendre del que fas tu.

En resum, com que aquesta eina és molt potent i jo ja hi havia treballat anteriorment, vaig decidir que per dur a terme la part pràctica l'utilitzaria. S'ha de dir que no l'utilitzaria només per poder treballar de manera més

[9] Vegeu: https://git-scm.com/.
[10] Vegeu: https://github.com/.

ordenada, sinó que com un dels meus objectius és fer una contribució a la comunitat en l'àmbit de la lingüística computacional en català, GitHub em permetrà fer el meu treball públic d'una manera molt fàcil.

PART PRÀCTICA

PRIMERA FASE: Procés de crear el *software* que pronominalitza

3. Cronologia i evolució de la creació del programa

En aquest apartat s'explicarà el procés cronològic de creació del programa, però no s'indagarà en temes complicats de programació.

3.1 Inici de la recerca

En el moment que vaig tenir el tema de la recerca decidit, em vaig posar a pensar com podria dur a terme la investigació. La meva primera idea va ser entrenar un model que es dediqués a detectar complements verbals. Aquesta idea jo la veia factible, ja que sabia que existia un corpus anomenat Ancora[11] que tenia els complements etiquetats tal i com jo els volia. El problema era que jo no tenia accés al corpus, i tampoc tenia els coneixements suficients per entrenar un model d'aquestes magnituds, tot i que no era el meu primer contacte amb IA, perquè havia realitzat alguns projectes de forma autodidacta.

Veient els grans problemes evidents, vaig decidir posar-me amb contacte amb el Barcelona Supercomputing Center (BSC), on hi estava cursant el programa Bojos per la Supercomputació. Cal remarcar que, sense la seva gran ajuda i atenció en tot moment, aquest treball no hauria estat possible.

En la nostra primera reunió el dia 4 d'abril, vaig exposar tant la meva idea, com el que havia pensat per dur a terme el treball. Ells em van explicar que recentment havien estat desenvolupant un model que feia exactament el que jo volia, entre d'altres coses. També em van donar accés a l'Ancora, em van explicar com les dades estaven estructurades i em van donar les eines necessàries per tractar-les en cas que jo ho necessités.

Una altra cosa que ens vam plantejar va ser si a partir del *software* pronominalitzador que jo podia crear utilitzant l'Spacy, podríem crear un

[11] Vegeu: http://clic.ub.edu/corpus/es/ancora.

corpus amb moltes oracions pronominalitzades, encara inexistent, que podria servir per a futures investigacions.

Davant d'aquesta fructífera reunió, em vaig posar a treballar pensant que crear el *software* que pronominalitzés seria bufar i fer ampolles.

3.2 Recerca encaminada

3.2.1 Instal·lació d'Spacy

Tot i que semblava que hauria de ser molt fàcil descarregar l'Spacy a l'ordinador de manera local, vaig tenir múltiples problemes.

El primer problema que va sorgir va ser que la versió que jo tenia instal·lada de Python era incompatible amb el model instal·lat d'Spacy. Segons la meva opinió i la meva experiència, un dels defectes de Python és la gran varietat de versions i problemes que hi ha entre elles, ja que, quan es treballa amb mòduls, és complicat que tot funcioni correctament i, depenent de la versió, el resultat canvia. En canvi, en un projecte en el qual s'utilitza el llenguatge c++, per exemple, la instal·lació pot ser complicada, però el resultat que s'obté no tindrà el caire aleatori que pot tenir a vegades Python.

Després de tres setmanes d'intents infructuosos, quan a la fi vaig aconseguir que l'Spacy funcionés, utilitzant el gestor de paquets Conda, vaig optar per començar a programar el projecte amb l'entorn de JupyterLab. Tot i que jo mai havia treballat amb un sistema de llibretes com el de Jupyter, em va semblar una molt bona opció per fer aquest projecte.

3.2.2 Analitzant com funciona l'Spacy

Quan vaig poder utilitzar l'Spacy, em vaig llegir les parts de la documentació que m'interessaven per tenir clar el potencial que podia extreure a aquesta eina tan potent.

Seguidament, em vaig posar a programar el complement directe determinat, però vaig veure que havia de veure exactament com el mòdul detectava els complements, ja que programar-ho m'estava costant més de l'esperat.

Cal remarcar que quan es va entrenar el model, es va fer una adaptació de l'anotació comuna que tenim per anomenar els complements, i es van dir tots seguint les *global dependencies,*[12] que acaba sent el mateix, però dit diferent.

Per fer aquest anàlisi, vaig optar per fer ús de la llibreria Pandas,[13] ja que em permetia fer taules i veure les dades d'una manera entenedora. Llavors, el que vaig fer va ser escriure moltes oracions de tots els tipus de complements, analitzar-les amb l'Spacy i després posar-les en una taula com la que es veu a continuació, per veure el resultat d'una manera més entenedora.

	0	1	2	3	4	5	6	7
0	Ves | ROOT	a | mark	comprar | xcomp	les | det	pomes | obj	al | case	mercat | obl	----
1	La | det	nena | nsubj	va | ROOT	a | mark	comprar | xcomp	aquest | det	vestit | obj	vermell | amod
2	El | det	nen | nsubj	compra | ROOT	el | det	cotxe | obj	verd | amod	----	----
3	En | det	Marti | nsubj	portara | ROOT	els | det	llibres | obj	a | case	la | det	Julia | obj
4	El | det	gerent | nsubj	portar | ROOT	aquestes | det	resolucions | obj	al | case	gabinet | obj	juridic | amod

Il·lustració 4. Taula amb oracions analitzades

Aquesta taula és molt més extensa, però amb quatre exemples ja es veu com funciona.

De l'estudi que vaig fer d'aquesta taula en vaig extreure la conclusió que generalment l'Spacy detecta els complements de manera correcta, però els etiqueta malament. Això és un greu problema, ja que, perquè funcioni correctament, s'ha d'estudiar com és cada complement detectat, i després s'haurà de programar perquè funcioni amb tots aquells patrons.

Veient que se'm girava molta més feina de l'esperada, vaig decidir tractar cada complement de forma meticulosa.

A l'apartat següent veurem quins problemes i solucions han sorgit durant tot el procés.

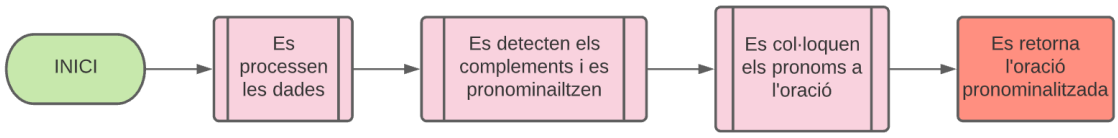
[12] Vegeu: https://universaldependencies.org/ca/index.html.
[13] Vegeu: https://pandas.pydata.org/.

4. Funcionament del programa, problemes i solucions

En aquest apartat s'explicarà el *software* encarregat de pronominalitzar. Per fer-ho, s'explicarà l'esquelet del programa i després ja s'aprofundirà en les parts més importants. Malauradament, no es pot explicar tot el codi per qüestions d'extensió, però el podeu consultar a: https://github.com/joangoma/PronomsTDR.

4.1 Estructura general i constants del programa

Aquest programa està estructurat en tres parts força marcades: processament de les dades, detecció dels complements i col·locació dels pronoms correctament a les oracions, tal i com es veu a la *il·lustració 5*.



Il·lustració 5. Esquelet del programa

Al tractar-se d'un codi d'una longitud extensa, vaig decidir estructurar-lo molt bé amb funcions perquè, quan tingués problemes, fos relativament senzill detectar-los i solucionar-los.

Abans d'explicar com funciona el codi, cal conèixer les variables constants del programa que aniran apareixent per tot el codi. És per això que cal conèixer exactament què representen. Com que aquestes variables són força extenses i si les posés senceres ocuparien 5 pàgines, només explicaré els valors que guarden i com es guarden. En el moment en què se'n faci ús, ja s'explicarà com s'utilitzen.

En primer lloc, hi ha un diccionari definit amb el nom de **PRONOMS,** que té guardat cada pronom amb les seves quatre formes corresponents: reforçada, elidida, plena i reduïda.

Tot i que els diccionaris no s'han explicat a l'apartat de teoria, ja que són un element una mica més complex que la resta, només cal saber que permeten guardar i accedir a dades amb uns costos logarítmics molt baixos, comparats amb les llistes, per exemple. És per això que he decidit utilitzar-los en certes constants, com la dels pronoms.

La variable PRONOMS es representa de la següent manera:

```
PRONOMS = {
    'em' : ['em', "m'", '-me', "'m"],
    'et' : ['et', "t'", '-te', "'t"],
    'es' : ['es', "s'", '-se', "'s"],
    'en' : ['en', "n'", '-ne', "'n"],
    'el' : ['el', "l'", '-el', "'l"],
    'la' : ['la', "l'", '-la', "-la"],
    'ens' : ['ens', "ens", '-nos', "'ns"],
    'us' : ['us', "us", '-vos', "-us"],
    'els' : ['els', "els", '-los', "'ls"],
    'les' : ['les', "les", '-les', "-les"],
    'li' : ['li', "li", '-li', "-li"],
    'ho' : ['ho', "ho", '-ho', "-ho"],
    'hi' : ['hi', "hi", '-hi', "-hi"]
}
```

Seguidament, hi ha una llista, anomenada **VERBS_CP**, que té emmagatzemats els verbs que fan una excepció amb el complement predicatiu. Tot i que aquesta llista no és gaire extensa, s'utilitza en diverses funcions, i per no repetir codi, vaig decidir fer-la una constant global.

Aquesta variable es representa de la següent manera:

```
VERBS_CP = ["FER", "DIR", "ELEGIR", "NOMENAR"]
```

A continuació, hi ha un altre diccionari, anomenat **VERBS_CONJUGATS,** que té guardades totes les formes verbals de certs verbs, que jo he considerat importants per pronominalitzar els complements correctament. Aquesta variable és força extensa, i per tant, només n'ensenyaré un tros. Cal dir que aquesta variable ha sigut necessària, ja que en certs casos l'Spacy no em proporcionava tota la informació que necessitava.

Aquesta variable es representa de la següent manera:

```
VERBS_CONJUGATS = {'SEMBLAR': ['semblo', 'sembles', ...],
                   "ELEGIR" : ['elegeixo', 'elegeixes', ...],
                   "DIR": ['dic', 'dius', ...],
                   "FER": ['faig', 'fas', ...],
                   "NOMENAR": ['nomeno', 'nomenes', ...]
                   }
```

Per últim, hi ha una llista, anomenada **PRON_BINARIES**, que representa la il·lustració 6 de combinacions binàries, explicada en l'apartat de teoria. Tot i que el contingut d'aquesta taula no és molt interessant, serà clau per poder pronominalitzar combinacions binàries.

Aquesta variable està representada de la següent manera:

```
PRON_BINARIES = [
[["s'hi"], ["-s'hi"]], [["se'n", "se n'"], ... ],
[["us hi"], ["-vos-hi", "-us-hi"]], ...],
[["m'hi"], ["-m'hi"]], ... ],
[["ens hi"], ["-nos hi", "'ns-hi"]], ...],
[["li hi"], ["-li-hi"]], ... ],
[["els hi"], ["-los hi", "'ls-hi"]], ...],
[["l'hi"], ["-l'hi"]], ...],
[["la hi"], ["-la-hi"]], ...]
]
```

Recordeu que si voleu veure el contingut sencer d'aquestes variables, podeu fer-ho accedint al meu perfil de GitHub.

4.2 Processament de dades

El procés que segueix el programa en aquesta primera fase és relativament senzill. Tal i com veurem en el diagrama de flux al final d'aquest apartat, el que es fa és el següent.

Primer de tot es llegeix l'input de l'usuari. Seguidament es "tokenitza" el text introduït per l'usuari i es va analitzant. En el cas que es detecti algun complement, es comprovarà de quin tipus és i com es pronominalitza, respectivament. En el cas que la paraula detectada sigui un pronom, aquest es guarda, ja que en el possible cas que després aparegui un complement, el complement i el pronom es puguin pronominalitzar alhora. Per exemple, això passaria amb l'oració: **Li** donem **molts llibres** => **Li'n** donem molts.

Cal remarcar que si l'Spacy identifica erròniament un pronom feble, com per exemple un article davant d'un nom, no es pot fer res perquè el resultat sigui correcte. Per tant, depenent de la precisió que tingui el model, el percentatge d'encert a l'hora de pronominalitzar serà més bo o més dolent.

A continuació s'explicarà amb deteniment la funció encarregada de fer aquesta tasca. Al principi d'aquesta funció es declaren totes les variables que es necessitaran perquè funcioni.

Primer de tot, hi ha una llista amb les etiquetes que poden etiquetar un complement predicatiu. Aquí es pot fer una petita observació, i és que aquestes etiquetes no es corresponen a l'etiqueta que hauria de tenir un complement predicatiu. Per tant, s'haurà de programar bastant més, com ja es veurà en l'apartat corresponent, perquè el complement predicatiu funcioni correctament.

La llista de nom **l** s'encarregarà de guardar la informació dels complements i la seva respectiva informació. Cada complement es representarà amb una llista que tindria la següent forma: [tipus de complement, pronom, [dependències], [nucli]]. Per exemple, si estigués representant el complement directe de l'oració: La nena compra **tres patates**, la llista l, tindria els següents valors: **['cdIndet', 'en', [tres], [patates]]**.

La variable de nom **t** s'encarregarà de tenir controlat si s'ha pronominalitzat aquell complement en la iteració corresponent.

El codi corresponent és el següent:

```
def main(s):
    cpp = ['amod', 'appos', 'nsubj']
    l = []
    t = True
```

Quan totes les variables estan declarades, es comença a iterar per la frase *"tokenitzada"*. Just al començament de cada iteració, hi ha una llista de nom **child,** que té totes les dependències d'aquella paraula. Aquesta llista és força important per poder extreure més informació necessària més endavant. Després, mitjançant una funció de nom **possible_complement()**, que no s'explicarà amb detall, es determina si aquell complement interessa, i sobretot es descarta l'opció que sigui un complement de nom. En el cas que la paraula que s'està iterant sigui una potencial candidata per ser un complement verbal, poden passar dues coses.

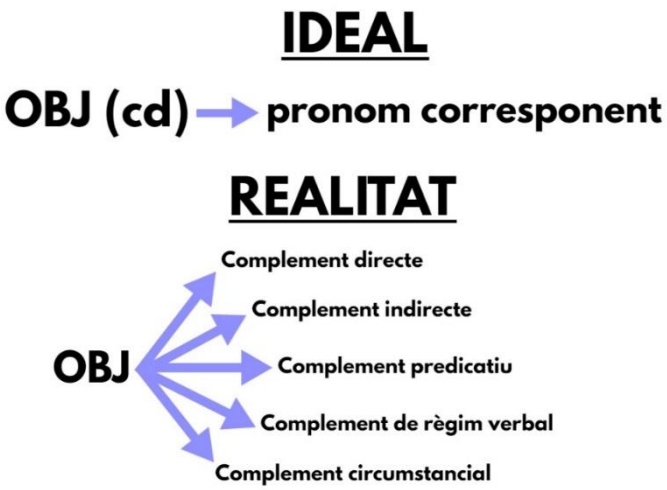
La primera opció, que és la que podem veure en el segment de codi de més a baix, contempla la possibilitat que la paraula analitzada en qüestió sigui un

pronom feble. Per determinar-ho, s'iterarà en el diccionari constant de nom **PRONOMS**, i, en cas de trobar alguna coincidència, es guardarà aquest pronom a la llista **l** amb el format comentat anteriorment.

```
    for i, token in enumerate(tkora):

        child = [child for child in token.children]

        if possible_complement(nlp(s), token) == True and t == True:
            for e in PRONOMS.keys():
                if str(token) in PRONOMS[e]:
                    l.append(['pron', str(token), child, [token]])
                    t = False
```

En el cas que no es tracti d'un pronom feble, el procés que es seguirà és una mica més llarg.

Tal i com es veu en el fragment de codi de la pàgina següent, el que es farà és comprovar de quin tipus és la dependència i derivar a la funció corresponent perquè determini el pronom adequat a l'hora de pronominalitzar. El procés, però, és una mica més complex, ja que tal i com s'ha comentat anteriorment, l'Spacy no etiqueta correctament els complements. Per tant, tot i que teòricament sempre que es detecta una dependència de nom **"obj"** hauria de ser un complement directe, en aquest model de l'Spacy pot ser múltiples coses, tal i com es veu en la il·lustració següent.



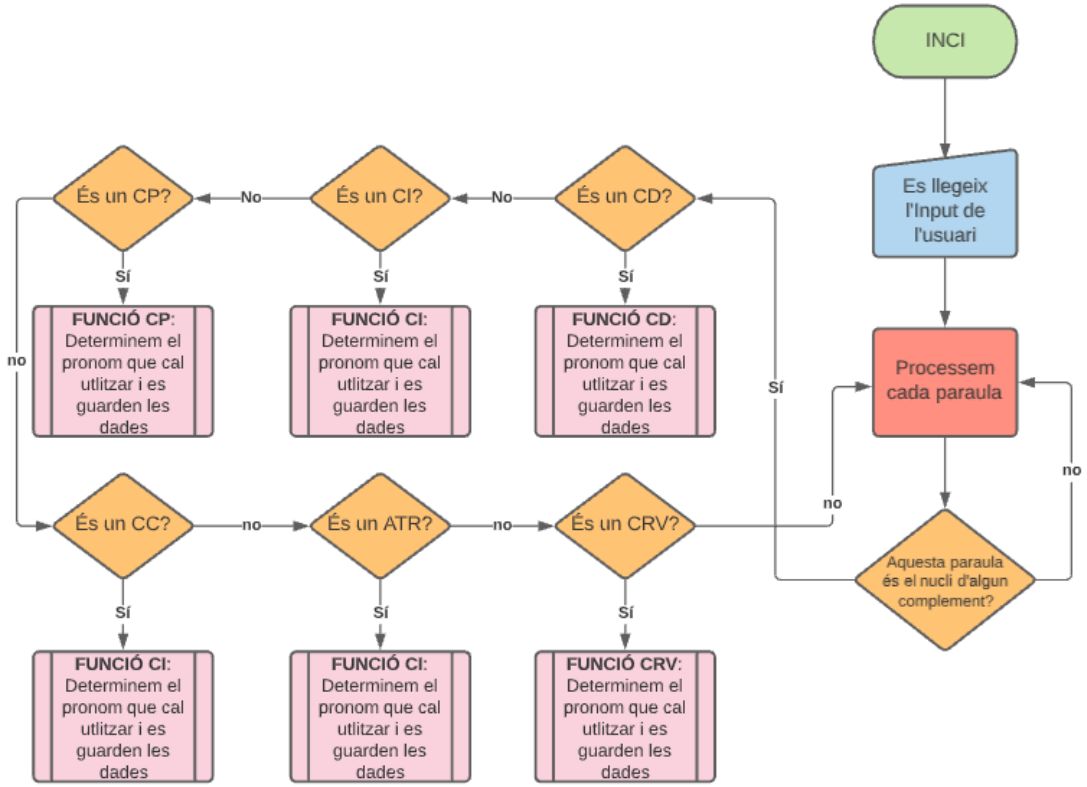
Il·lustració 7. Com funcionen les etiquetes

Aquest procés de comprovar quin complement està representant l'etiqueta **obj** el fa la funció **possible_complement_obj()**, que després deriva a les funcions particulars, que ja determinen el pronom adequat per substituir aquell complement.

Els altres condicionals comproven si aquell complement es tracta d'un atribut, d'un complement predicatiu, o d'un complement circumstancial. Les peculiaritats de cada complement s'explicaran amb deteniment quan s'expliqui exclusivament aquell complement. Cal recordar que tot aquest fragment de codi està a dins del bucle **for**. Per tant, s'executa per cada paraula que té l'oració.

```
if t == True:
    if str(token.dep_) == 'obj' and t == True:
        el = possiblitat_comp_obj(str(token), token, child, s, tkora)
        if el != []:
            l.append(el)
            print(1, l)
            t = False

    elif str(token.dep_) == 'cop' and t == True:
        el = atribut(str(token), token, child, tkora, False)
        if el != []:
            l.append(el)
            t = False

    elif str(token.dep_) == 'ROOT' and atribut_semblar(nlp(s), token) and t == True:
        el = atribut(str(token), token, child, tkora, True)
        if el != []:
            l.append(el)
            t = False

    elif str(token.dep_) in cpp and t == True:
        el = complement_predicatiu(token, child, tkora)
        if el != []:
            l.append(el)
            print(l)
            t = False

    elif str(token.dep_) == "obl" and t == True:
        el = complement_indirecte(token, child, s) + [child]
        if el != []:
            l.append(el)
            t = False

if t == False and l[-1][0] != 'pron':
    l[-1].append([token])
```

En el següent diagrama de flux es pot veure aquest procés simplificat.



Il·lustració 8. Processament de dades

4.3 Detecció de complements

En aquest apartat s'explicarà en profunditat com funciona el codi que realment detecta i tria el pronom adequat per pronominalitzar el pronom corresponent. L'explicació seguirà l'estructura del codi. És a dir, per complements, com es veu en el diagrama de flux "Processament de dades". Cada complement es dividirà en diversos subapartats i s'il·lustrarà amb el codi corresponent.

4.3.1 Complement directe

La funció del complement directe és una de les més senzilles ja que, tot i que aquest complement es pot pronominalitzar de tres maneres, el model de l'Spacy els sol etiquetar força bé i no hi ha tantes possibilitats com en altres complements.

Just a l'inici, es defineix la funció de nom **complement_directe()**, que rep per paràmetres el nucli del complement "tokenitzat" de nom **tkpar**, les

dependències del complement de nom **dep**, i l'oració en qüestió "tokenitzada" de nom **tkora**.

Al començament de la funció, es definiran un seguit de variables que emmagatzemaran en format de llista certes dades que seran molt importants. Aquestes dades es poden separar en quatre blocs.

El primer bloc hi haurà les dades necessàries per pronominalitzar el complement directe determinat. Aquestes dades són les següents: **cdDet**, que guarda totes les cadenes amb les quals un complement directe determinat pot ser introduït; **proDet**, que guarda tots els possibles pronoms que poden substituir el complement directe determinat; i per últim, la llista **demostr**, que guarda tots els determinants demostratius.

En el segon bloc es troben les variables que són necessàries per detectar i pronominalitzar correctament el complement directe indeterminat. Aquestes són les següents: **quant**, que guarda tots els quantitatius que té el català; i la llista de nom **indef**, que guarda tots els articles indefinits.

En el tercer bloc es troba la variable de nom **cdNeut**, que senzillament emmagatzema les paraules "això" i "allò", les quals introdueixen el complement directe neutre.

Per acabar, hi ha dues llistes que emmagatzemen dades que seran útils per descartar la possibilitat que el complement que s'està analitzant no sigui un complement directe. Recordeu que l'Spacy no etiqueta gaire bé els complements.

Fins ara, el codi es veurà de la següent manera:

```
def complement_directe(tkpar, dep, tkora):

    cdDet = ["el", "la", "els", "les", "l'", "aquest", "aquests", ...]
    proDet = ["el", "la", "els", "les", "l'"]
    demostr = ["aquest", "aquesta", "aquests", "aquestes", ...]

    quant = ["massa", "força", "prou", "més", "menys", "gens",
             "bastant", "bastants", ...]
    indef = ["un", "una", "uns", "unes"]

    cdNeut = ["això", "allò"]

    prp = ['a', 'per']
    l = ["FER", "DIR", "ELEGIR", "NOMENAR"]
```

A continuació es defineix una variable de tipus cadena de nom **verb**, que guardarà el verb principal de l'oració. Com que l'Spacy separa l'oració per paraules, si el verb és compost, no l'identificarà completament. Per exemple, si el verb de l'oració fos *ha comprat*, detectaria que el verb era *comprat*, no *ha comprat*. Per solucionar aquest problema, s'itera per tota l'oració i, tant si troba un verb auxiliar com un verb principal, l'afegeix a la variable verb. D'aquesta manera, es podrà determinar, en funció de si el verb és compost o no, si el pronom que substitueix el complement s'ha d'apostrofar o no.

Seguidament, amb un parell de condicionals es comprova que el complement que s'està analitzant no sigui ni un complement indirecte ni un complement predicatiu. Per fer això, es comprova que el verb de l'oració no pertanyi a cap conjugació dels verbs "**fer-se, dir-se, elegir, nomenar**", d'aquí la importància del diccionari **VERBS_CONJUGATS** explicada abans. Per comprovar que no sigui un complement indirecte, es mira com està introduït el complement. En el cas que alguna d'aquestes condicions es compleixi, la funció deixarà d'executar-se.

El codi que s'haurà afegit a la funció serà el següent:

```
    verb = ""
    for token in tkora:
        if str(token.dep_) == "aux": verb += str(token) + " "
        if str(token.dep_) == "ROOT": verb += str(token)

    for e in l:
        if verb in VERBS_CONJUGATS[e] and tkpar.pos_ == "NOUN": return []

    if str(tkpar) in prp: return []
```

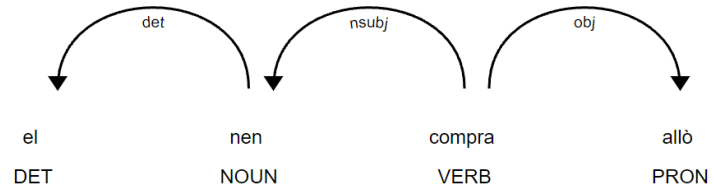
4.3.1.1 Complement directe neutre

Ara que ja es tenen totes les dades necessàries, cal començar a programar els casos concrets del complement directe. Per fer això, prèviament cal veure com l'Spacy analitza cada cas.

El cas del complement directe neutre potser és un dels més fàcils de pronominalitzar correctament per un humà, però veurem que no passa el mateix a l'hora de programar.

Tal i com es va explicar en el marc teòric, un complement directe neutre pot estar introduït per *això*, *allò* o una oració subordinada substantiva.

El primer cas, que seria un complement introduït per *això* o *allò*, és relativament senzill de programar, ja que només s'ha de comprovar el primer element de les dependències que té el nucli. Vegem com l'Spacy analitza una oració d'aquest tipus a la imatge següent.



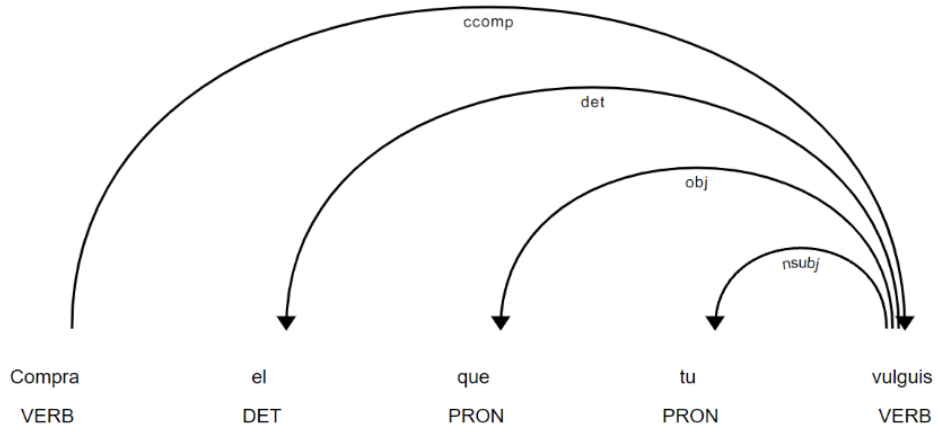
Il·lustració 9. Oració analitzada CD neutre

Tal i com es veu en aquest exemple i tots els altres que he estudiat, per determinar si és un complement directe neutre d'aquest tipus, només s'haurà de mirar l'element que encapçala les dependències.

Per programar aquesta norma, s'hauria de comprovar si el nucli del complement directe està inclòs en la llista de nom **cdNeut** descrita anteriorment. Per tant, al codi se li haurà d'afegir la línia següent:

```
if str(tkpar) in cdNeut: return ['cdNeut', 'ho']
```

En canvi, el segon cas és molt més complicat de programar, ja que una oració subordinada substantiva pot aparèixer de moltes maneres. Per tant, si no es pot comprovar cada possibilitat, com es feia en el primer cas, s'hauria de buscar algun patró de com l'Spacy analitza aquestes oracions. Vegem-ne un exemple:



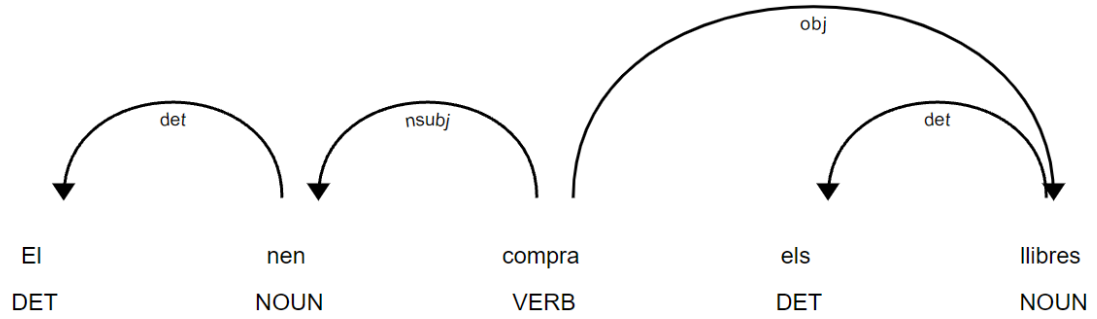
Il·lustració 10. Oració analitzada CD neutre oració subordinada substantiva

Tal i com havia fet en l'altre cas, vaig intentar buscar alguna similitud en la manera d'analitzar les oracions subordinades, però tal i com he comentat anteriorment, hi havia moltes possibilitats. Tot i que vaig intentar programar que aquest tipus de complement directe es pogués pronominalitzar, vaig veure que estava invertint molt temps per poc progrés. Per tant, vaig decidir que el programa només analitzaria oracions simples. Tot i això, no descarto en un futur implementar aquesta funcionalitat.

4.3.1.2 Complement directe determinat

El següent cas és el complement directe determinat. Tal i com he fet en el complement anterior, la metodologia serà estudiar com l'Spacy analitza les oracions i veure les normes que s'han d'aplicar.

Per veure exactament el que s'està buscant quan es programa, vegem l'anàlisi morfosintàctica que faria l'Spacy davant l'oració "El nen compra els llibres" amb la imatge següent.



Il·lustració 11. Oració analitzada CD determinat

A partir d'aquesta estructura i d'analitzar uns quants exemples més, es pot deduir què és important a l'hora de pronominalitzar el complement directe.

Tal i com es va explicar en l'apartat de teoria corresponent, el complement directe determinat pot estar introduït per un article determinat o un determinant demostratiu. Llavors, es pot treure la conclusió que sempre que l'Spacy etiqueti un complement com a "**obj**", només caldrà mirar com està introduït. Si està a dins d'uns dels casos descrits anteriorment, es pronominalitzarà amb el pronom corresponent.

Tot i que això és cert, per tal que el programa funcioni correctament, cal tenir en compte una petita excepció, que és quan tenim un complement directe introduït per un apòstrof. En aquesta situació, s'haurà de saber el gènere de la paraula per tal d'utilitzar el pronom correcte a l'hora de pronominalitzar-lo.

Per tant, per traduir aquestes normes a codi, el primer que s'haurà de comprovar serà mirar si l'element que encapçala la llista de les dependències està inclòs a la llista **cdDet**. En cas que aquest condicional avaluï l'expressió

com a certa, se separarà l'oració en una llista de nom **l**, on cada element serà una paraula. Per exemple, si s'estigués analitzant l'oració: *El nen compra l'ordinador*, aquesta llista seria així: ["El", "nen", "compra", "l", "ordinador"]. El codi que duria a terme aqueta tasca seria el següent:

```
        if str(dep[0]) in cdDet:
            sig = ["'", "-"]
            l = ora.split()

            for e in l:
                for s in sig:
                    if s in e:
                        l.remove(e)
```

Amb aquesta llista, puc accedir a molta més informació de l'oració de manera senzilla.

Abans de determinar quin pronom és l'adequat, es comprova que la paraula anterior a la primera que introdueix les dependències no sigui ni **a**, ni **per**, ja que llavors no es tractaria d'un complement directe. Aquest condicional es programa de la següent manera:

```
if l[l.index(str(tkpar))-len(dep)-1] in prp: return []
```

Aquesta condició és necessària en cas que l'Spacy hagi analitzat alguna oració de forma incorrecta. En el cas que aquesta condició hagi sigut avaluada com a falsa, només caldrà comprovar si la paraula que introdueix l'oració és un demostratiu o un determinant.

Al primer cas, s'haurien de fer dues comprovacions. Si es tracta d'un demostratiu que es presenta en forma plural, el pronom corresponent també ho haurà d'estar; i si està en singular, s'haurà de pronominalitzar amb un pronom singular. Tot i que tal i com està programat pot semblar una mica confús, cal tenir clar de quines llistes es parteix i com funcionen els índexs a programació.

Aquests condicionals es programarien de la següent manera:

```
if str(dep[0]) in demostr:
      if demostr.index(str(dep[0])) > 3:
           return ['cdDet', proDet[demostr.index(str(dep[0]))-4]]
      else:
           return ['cdDet', proDet[demostr.index(str(dep[0]))]]
```

L'altre cas general que es pot trobar és si la paraula que introdueix les dependències és un article definit. En aquest cas, tot i ser una mica més fàcil que l'altre, s'ha de contemplar el problema de l'apostrofació i hi hauran dos condicionals.

El primer comprovarà si l'article en qüestió està apostrofat. En cas d'estar-ho, es retornarà l'article corresponent segons el gènere de la paraula que acompanya. Per fer això, es farà ús d'una funció auxiliar de nom **article_apostrofat_segons_gènere()**. Per temes d'extensió, aquesta funció no s'explicarà.

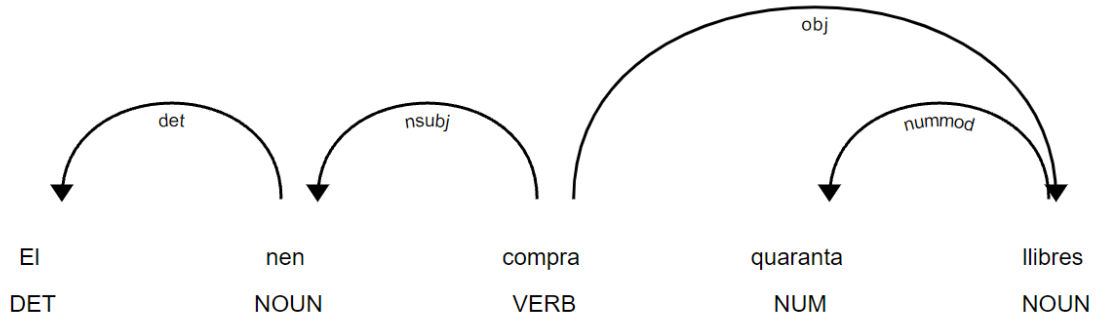
L'altre cas que queda serà retornar el pronom en la forma que es trobava originalment. Totes aquestes condicions es programaran de la manera següent:

```
else:
      if str(dep[0]) == "l'" or str(dep[0]) == "L'":
           return ['cdDet', article_apostrofat_segons_genere(str(dep[0]), tkora)]
      else:
           return ['cdDet', str(dep[0])
```

4.3.1.3 Complement directe indeterminat

L'últim cas que presenta el complement directe és l'indeterminat. Tal i com s'ha fet en els dos últims apartats, a continuació es veurà com l'Spacy analitza aquest tipus de complements.

Per veure les normes que s'estan buscant en aquet tipus de complement, vegem com s'analitza l'oració "El nen compra quaranta pomes".



Il·lustració 12. Oració analitzada CD indeterminat

Recordeu que el complement directe indeterminat pot estar introduït per un determinant quantitatiu, un determinant indefinit, un numeral o res.

Veient l'estructura que es genera de cada tipus de complement directe indeterminat, vaig arribar a les següents conclusions: com que els quantitatius i els indefinits són una sèrie de paraules finites, vaig pensar que podia detectar aquest tipus de complement si l'Spacy l'etiquetava com a "**obj**", i la paraula que l'introduïa era una de les que he anomenat anteriorment. Per tant, per programar aquestes condicions vaig utilitzar un parell de condicionals i, fent ús les llistes explicades a l'inici d'aquest apartat de nom **indef** i **quant**, en vaig tenir prou. Traduït a codi, es programarà així:

```
    if str(tkpar.pos_) in morph and str(dep[0]) in indef: t = 1
    if str(tkpar.pos_) in morph and str(dep[0]) in quant: t = 1
```

El següent cas, el dels numerals, tot i ser una mica diferent, l'únic que necessitava saber era si aquella paraula era un numeral. Aquest cas, però, no el podia resoldre tal i com ho havia fet anteriorment, ja que els numerals són infinits, i el programa havia de funcionar tant si "la nena comprava tres pastissos", com si "la nena en comprava quaranta". Per solucionar aquest problema, l'Spacy té una etiqueta de nom **pos_**, que retorna el tipus morfològic de la paraula. D'aquesta manera, si em retornava l'etiqueta **NUM**,[14] sabia que es tractava d'un numeral. Tot i que aquesta solució és força bona, vaig observar que no funcionava correctament amb nombres de dues paraules o més paraules o amb guionets, com "vint-i-set". Com que hauria

[14] Vegeu: https://universaldependencies.org/u/pos/NUM.html.

hagut de programar bastant més, vaig decidir no contemplar aquests casos, i en un futur intentaré solucionar aquest problema menor.

Per programar aquesta condició, vaig fer servir un altre condicional que comprovava el tipus de valor que tenia l'etiqueta **pos_**. El codi es programarà així:

```
if str(tkpar.pos_) == "NOUN" and str(dep[0].pos_) == 'NUM': t = 1
```

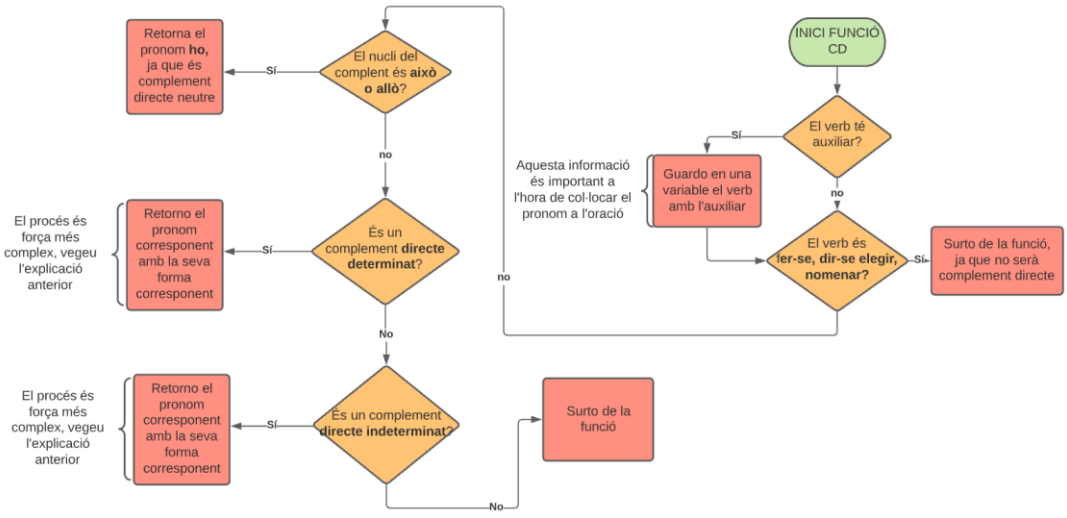
L'últim cas és quan el complement directe no està introduït per res: retornaríem el pronom corresponent i ja està. Aquesta condició funciona, perquè tots els altres complements que s'etiqueten com a **obj** sempre tenen alguna paraula que els introdueix. Amb una estructura molt semblant als condicionals anteriors, el codi es programarà així:

```
if str(tkpar.pos_) == "NOUN" and dep == []: t = 1
```

Fixeu-vos que en tots els condicionals es comprova que el nucli del complement sigui un nom, i de fet, cap condicional no retorna res. Això ho vaig fer per simplificar el codi, ja que haurien retornat tots el mateix. Llavors, si controlo si alguna d'aquestes expressions booleanes és certa mitjançant la variable **t**, ja en tinc prou. Al final de la funció afegeixo el codi que veureu a continuació i la funció del complement directe ja estaria acabada.

```
    if t == 1: return ['cdIndet', 'en']
```

El diagrama de flux d'aquesta funció és el següent:



Il·lustració 13. Diagrama de flux complement directe

4.3.2 Complement indirecte

La funció que s'encarrega de detectar i pronominalitzar el complement indirecte no és molt complexa, però veureu que, per mancances del model de l'Spacy, no es podrà aconseguir un percentatge d'encert tan alt com a la funció del complement directe.

Just a l'inici es defineix la funció de nom **complement_indirecte()**, que rep per paràmetres el nucli del complement "tokenitzat" amb el nom **tkpar**, les dependències del complement de nom **dep**, i per acabar, l'oració "tokenitzada" amb el nom de **tkora**.

A continuació, es defineix la variable que s'utilitza en tota la funció. Aquesta variable, que és la llista **prp**, conté les preposicions que poden introduir un complement indirecte. També conté les contraccions que es formen quan s'ajunta la preposició **a** amb l'article **el**. El codi desenvolupat és el següent:

```
def complement_indirecte(tkpar, dep, tkora):
    prp = ['a', 'per', 'al', 'als']
```

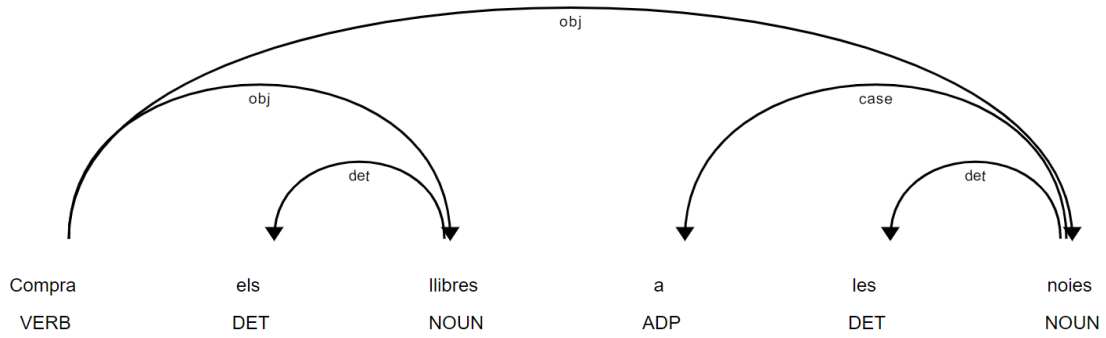
Tot seguit, tal i com es va fer a la funció del complement directe, se separa l'oració rebuda per paràmetres en una llista, per poder accedir a la informació concreta d'una manera més senzilla. Com que aquest codi ja es va explicar, aquí no el repetiré.

Una vegada ja es tenen totes les dades necessàries, primer de tot es comprova que el possible complement no sigui un atribut. Recordeu que l'Spacy no etiqueta bé els complements. Per fer aquesta comprovació, el que es fa és iterar per tota l'oració "tokenitzada" i mirar si hi ha algun complement amb la dependència **cop**, o un nucli del predicat que formi part de la conjugació verbal del verb *semblar*. Aquestes dues comprovacions es fan de la següent manera:

```
for e in tkora:
      if str(e.dep_) == "cop": return []
      elif str(e) in VERBS_CONJUGATS['SEMBLAR']: return []
```

Quan ja es tenen les dades que es necessiten i s'ha descartat la possibilitat que el complement sigui un atribut, comença la part del codi més complexa. Abans d'explicar directament el codi, s'ha de veure com l'Spacy analitza

aquest tipus d'oracions. Ho veurem amb l'exemple de l'oració: Compra els llibres per a les noies.



Il·lustració 14. Oració analitzada CI

Veient com analitza aquesta oració i moltes altres, vaig extreure les següents conclusions. Primer de tot, veiem que sempre que un complement estigui etiquetat com a **obj** o **obl**, i no sigui complement directe, atribut i estigui introduït per **a**, o **per a**, serà un complement indirecte. Cal recordar que, segons les *universal dependencies,* aquest complement hauria de ser etiquetat com a **iobj**. Com que l'Spacy no ho etiqueta bé, cal una programació més complexa.

També vaig observar que necessitava informació referent al número del nucli del complement que s'estava analitzant, ja que la forma del pronom varia.

Per últim, vaig veure que tots els complements circumstancials i de règim verbal que estiguessin introduïts per **a**, o **per a** no els podria detectar, ja que l'anàlisi que feia l'Spacy era exactament el mateix amb els tres casos. Per exemple, l'anàlisi de l'oració *Tots contribuïm a la causa* no es diferencia en res de l'anàlisi de l'oració *Truca a la nena,* excepte del significat. És per això que no tindré cap manera, de moment, de determinar si aquell complement es tracta d'un complement de règim verbal, d'un CI, o d'un CC.

Per poder pronominalitzar correctament aquests casos que no funcionen, el que es necessitaria seria tenir un model amb un percentatge d'encert més alt. Tot i que em vaig plantejar entrenar-ne un jo, em suposava una inversió de temps molt gran, i amb molt poques probabilitats que el meu model fos

millor que el que havien entrenat uns professionals. Per tant, ho vaig descartar.

Per programar els casos en els quals sí que es pot detectar i pronominalitzar el complement correctament, el primer que s'ha de fer és comprovar que les dependències no siguin nul·les, ja que un CI mai podria existir sense estar introduït. En el cas que aquesta expressió booleana sigui certa, es comprova, en el cas que l'Spacy hagués analitzat malament l'oració, que la paraula anterior a la primera dependència no estigui inclosa a la llista **prp**. En cas afirmatiu, es guarda el resultat en una variable de nom **t**. Aquesta excepció teòricament no s'hauria de contemplar, però vaig observar que en certes oracions el fenomen descrit anteriorment passava. També es comprova que la paraula que introdueix el complement sigui una preposició. En el cas que aquesta condició sigui falsa, es sortirà directament de la funció.

A continuació el que faig és guardar a una llista la informació morfològica que retorna l'Spacy de l'etiqueta **morph**. Si, per exemple, es mirés què retorna **morph** en la paraula **nen** de l'oració *El pare compra tres llapis pel nen*, l'etiqueta **morph** retornaria el següent: *Gender=Masc|Number=Sing*.

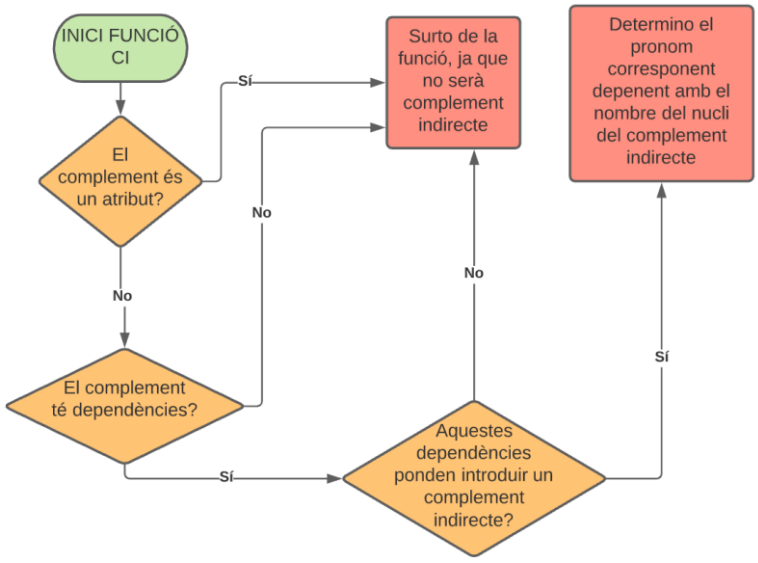
Per tant, tenint aquesta informació guardada, puc accedir al nombre i determinar la forma adequada del pronom.

Seguidament, en el cas que la paraula que introdueix el complement estigui inclosa en la llista **prp** o bé la variable **t** sigui certa, s'iterarà per la llista que ha guardat la informació de l'etiqueta **morph**. Quan es trobi la informació adient que es refereix al nombre del nucli, es comprovarà quin complement toca retornar. I amb un parell de condicionals, ja se sabrà el pronom adequat per substituir qualsevol complement indirecte.

El codi que realitza tot el procés descrit anteriorment és el següent:

```
    if len(dep) >= 1:
        if l.index(str(tkpar))-len(dep)-1 >= 0:
            if l[l.index(str(tkpar))-len(dep)-1] in prp: t = True
        if str(dep[0].dep_) != 'case': return[]

        l = str(tkpar.morph).split('|')

        if str(dep[0]) in prp or t:
            for e in l:
                ml = e.split('=')
                if ml[0] == 'Number':
                    if ml[-1] == 'Sing': return ['ci','li']
                    elif ml[-1] == 'Plur': return ['ci', 'els']
    return []
```

En el següent diagrama de flux es veu tot el procés descrit anteriorment d'una manera il·lustrada.



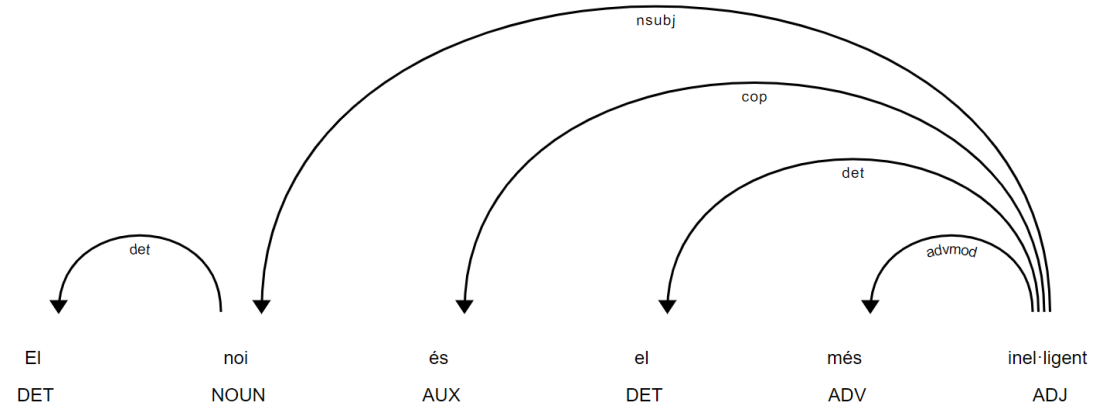
Il·lustració 15. Diagrama de flux CI

4.3.3 Atribut

La funció que s'encarrega de detectar i pronominalitzar correctament l'atribut potser és una de les més complexes per culpa de la diferència notable de com s'ha d'analitzar seguint les *universal dependencies*[15] i com s'analitza en català.

[15] Vegeu: https://universaldependencies.org/u/dep/cop.html.

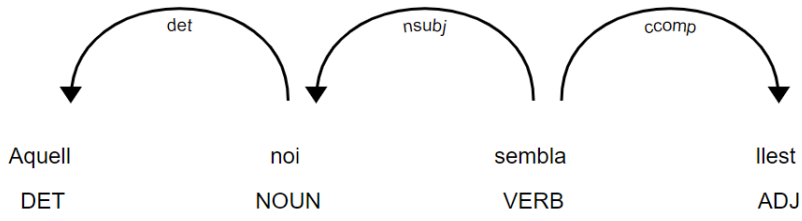
Com que la programació anirà totalment lligada a com analitza l'Spacy aquestes oracions, primer vegem-ne alguns exemples. Si s'analitzés l'oració *La noia és la més intel·ligent*, s'obtindria el següent:



Il·lustració 16. Oració analitzada atribut

Tal i com s'explica a la pàgina web de les *universal dependencies,* el nucli de l'atribut no és el verb, sinó la paraula que depèn d'ell. És per aquest fet que s'haurà de programar més per determinar les dependències d'aquest complement, ja que s'hi s'intenta accedir mitjançant l'etiqueta **dep_**, retornarà una llista buida.

L'altra norma que varia és que el verb *semblar*, que en català determina un atribut, per a l'Spacy és un verb predicatiu. És per això que s'haurà de fer *hard-code,* que consisteix a programar explícitament tots els casos d'un problema concret, per contemplar els casos on l'atribut està format pel verb *semblar*. Vegeu el següent exemple de l'oració *Aquell noi sembla llest*:



Il·lustració 17. Oració analitzada atribut

Vaig observar que, quan importa el significat, no es pronominalitza correctament. Per exemple, en l'oració *El nen és a València*, hi ha un complement circumstancial. En canvi, en l'oració *El nen està a la lluna de València*, el complement és un atribut. Passaria el mateix amb els verbs semicopulatius.

Just al començament es defineix la funció **atribut()**, que rep per paràmetres el verb que s'està analitzant "tokenitzat", de nom **tkpar**. També rep l'oració *"tokenitzada"* amb el nom de **tkora**, i per últim rep una variable booleana que emmagatzema si el verb amb el qual s'està tractant forma part de la conjugació verbal del verb *semblar*.

Just a continuació, es comprova si el verb amb el qual s'està operant forma part de la conjugació verbal del verb *semblar* mitjançant el paràmetre **semb**. En cas que aquesta condició sigui falsa, s'intentà identificar totes les dependències d'aquest complement. Això es farà iterant per totes les dependències del nucli del predicat, i sempre i quan aquestes siguin diferents del subjecte, les afegirà.

En el cas que l'avaluació de l'expressió sigui falsa, l'únic que s'ha d'emmagatzemar són les dependències del complement etiquetat com a **ccomp**. Això funciona perquè l'estructura de l'anàlisi que genera l'Spacy, amb oracions on el verb és *semblar* i s'està parlant d'un atribut, les dependències sempre estaran representades per l'etiqueta **ccomp**.

Quan ja es tenen les dependències, es crida a una funció externa, que comprova que realment s'hagin afegit totes. Aquesta tasca és important, ja que, perquè s'inclogui correctament el pronom en l'oració inicial, la llista que representa les dependències ha de ser correcte.

Tot el que s'ha explicat es programa així:

```
def atribut(tkpar, tkora, semb):

    if semb == False:
        for token in tkora:
            if token.dep_ == 'ROOT':
                l = [child for child in token.children]
                l1 = []
                for i, e in enumerate(l):
                    if str(e.dep_) != 'nsubj':
                        l1.append(e)
                l = l1
    else:
        for token in tkora:
            if token.dep_ == 'ROOT':
                l = [child for child in token.children]
                l1 = []
                for i, e in enumerate(l):
                    if str(e.dep_) == 'ccomp':
                        l1.append(e)
                        for c in e.children:
                            l1.append(c)

        l = l1

    dep = l
    dep = dependencies_completes(dep)
```

A continuació, en una variable booleana de nom **atr,** es guardarà si el verb de l'atribut que s'està pronominalitzant es tracta de *semblar* o bé *ser* i *estar*. Aquesta comprovació només es fa per tenir més informació sobre el complement més endavant.

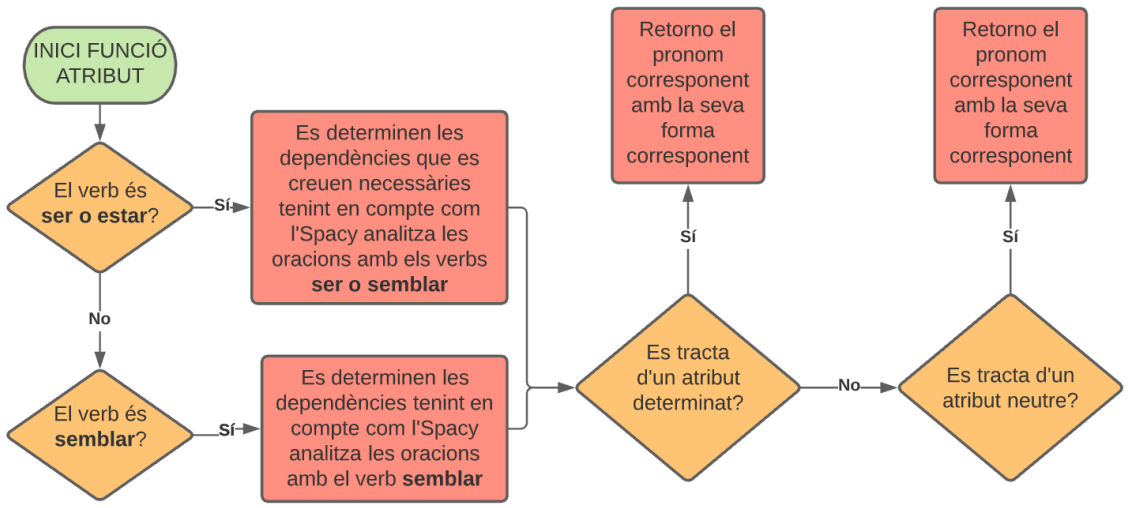
Seguidament, tal i com s'ha fet en les funcions del complement directe i indirecte, se separen totes les paraules que formen l'oració en una llista per tenir més informació sobre l'oració.

Una vegada es té això, es comprova si la paraula que introdueix l'atribut és un article determinat. En cas que aquesta comprovació sigui certa, mitjançant una estructura molt semblant a la que es va utilitzar en l'apartat del complement directe determinat, es retorna el pronom adient tenint en compte si l'article està apostrofat o no, i amb la petita diferència que no pot estar introduït per un demostratiu. En el cas que no sigui determinat, sempre es pronominalitza mitjançant el pronom **ho**. Per tant, en el cas que totes les condicions anteriors hagin sigut falses, es retornarà el pronom **ho**.

El codi que executa totes les condicions anterior és el següent:

```
if semb == True: atr = "atr1"
else : atr = "atr"

proDet = ["el", "la", "els", "les", "l'", "L'"]
ora = str(tkora)

sig = ["'", "-"]
l = ora.split()
for e in l:
    for s in sig:
        if s in e:
            l.remove(e)
            l += e.split(s)

p_1 = l.index(str(tkpar))+1 #posició de la següent paraula al verb

if str(tkora[p_1]) in proDet:
    if str(tkora[p_1]) == "l'" or str(tkora[p_1]) == "L'":
        return [atr, article_apostrofat_segons_genere(str(tkora[p_1]), tkora), dep]
        return [atr, str(tkora[p_1]), dep]
else:
    return [atr, 'ho', dep]
```

El diagrama de flux que representa tots els processos executats anteriorment és el següent:

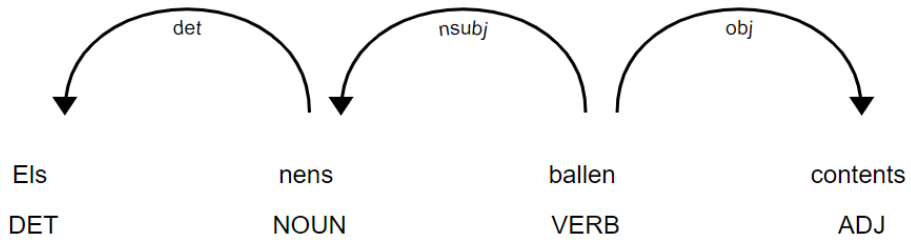


Il·lustració 18. Diagrama de flux atribut

4.3.4 Complement predicatiu

La funció del complement predicatiu és una funció que funciona força bé, però s'hauria de polir perquè acabés de funcionar en totes les excepcions.

Tal i com es va explicar en l'apartat corresponent de teoria, el complement predicatiu es pronominalitza per **hi**, menys quan està introduït per *fer-se, dir-se, elegir, nomenar,* que es pronominalitza per **en**. Per exemple, l'oració *els nens ballen contents* s'analitzaria així:



Il·lustració 19. Oració analitzada complement predicatiu

Com es pot veure, el complement predicatiu s'etiquetaria com a **obj** i sempre hauria de ser un adjectiu. Per tant, es podria fer una aproximació força bona comprovant el tipus de dependència i la categoria gramatical de la paraula. Això funciona perquè anteriorment ja s'ha comprovat que aquell complement no sigui ni un complement directe, ni un CI, ni un ATR.

És cert que faltaria comprovar que el complement predicatiu no estigués introduït per algun dels verbs que formen l'excepció, però és força complex programar-ho i en el seu moment vaig decidir deixar-ho a banda per poder fer un programa que funcionés en casos més generals, ja que era un problema bastant concret.

Hi ha certes oracions on el complement predicatiu no es pronominalitza, ja que l'Spacy l'etiqueta diferent i malauradament no n'hi ha prou amb comprovar l'etiqueta i les normes que he explicat anteriorment per determinar el complement i com es pronominalitza. Tot i que aquest problema amb l'ambigüitat de les etiquetes també passa amb el complement directe, en aquest cas tinc prou informació morfològica per determinar el tipus de complement. Per tant, la funció que analitza i pronominalitza el complement predicatiu està incompleta, ja que he prioritzat que funcioni en menys casos,

però que en aquests casos funcioni bé. La meva intenció és millorar-la en el futur.

Al principi es defineix la funció de nom **complement_predicatiu()**, que rep per paràmetres el nucli del complement "tokenitzat" amb el nom de **tkpar**, les dependències del complement amb el nom de **dep**, i l'oració *"tokenitzada"* amb el nom de **tkora**.

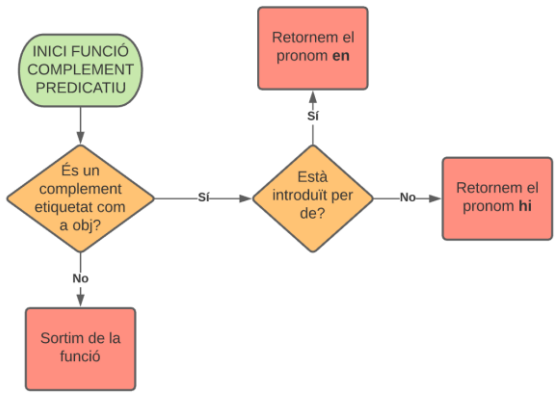
A continuació, com que només s'ha de comprovar la primera norma esmentada abans, el que s'ha de programar és molt senzill en comparació a les altres funcions: amb dos condicionals cal comprovar si la paraula és un adjectiu, i en el cas que ho sigui comprovar si està introduïda per la preposició **de**. En el cas que aquesta comprovació sigui certa, es retornarà el pronom **en** de la forma corresponent. En el cas que això no sigui cert, es retornarà el pronom **hi** i es sortirà de la funció.

Tot i que aquesta funció pot semblar molt simple, si s'estiguessin incloent les excepcions, seria una de les més complicades.

Tot el que s'ha explicat es programa de la manera següent:

```
def complement_predicatiu(tkpar, dep, tkora):

    if str(tkpar.pos_) == 'ADJ' and len(dep) > 0:
        if str(dep[0]) == 'de' or str(dep[0]) == 'd'': return ['cp',
'en']
    if str(tkpar.pos_) == 'ADJ':
        return ['cp','hi']



    return []
```

Aquest simple procés es veu il·lustrat en el següent diagrama de flux.

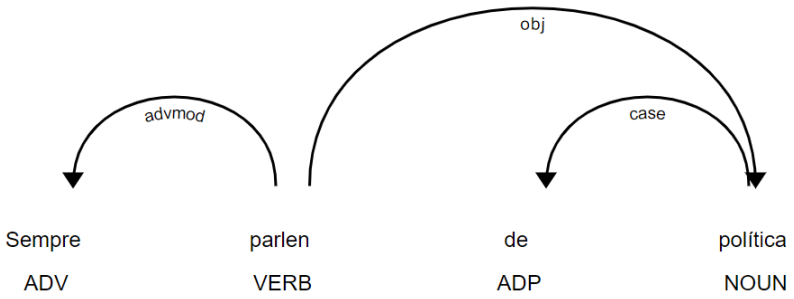


Il·lustració 20. Diagrama de flux complement predicatiu

4.3.5 Complement de règim verbal

La funció que s'encarrega de pronominalitzar i substituir aquest complement utilitza una aproximació força bona, però que es quedada limitada per la poca precisió que retorna l'Spacy.

Tal i com es va explicar en l'apartat corresponent de teoria, el complement de règim verbal, sempre que estigui introduït per **de**, es pronominalitzarà amb el pronom **en**, i els altres casos es pronominalitzarà amb el pronom **hi**. Vegem un exemple de com l'Spacy analitza aquest tipus de complements amb l'oració *Sempre parlen de política.*



Il·lustració 21. Oració analitzada CRV

Veient aquest exemple i uns quants més, es pot arribar a la conclusió general que quan aquest complement estigui introduït per **de** es pronominalitzarà d'una manera i quan estigui introduït per **a, en, amb, per**, es pronominalitzarà d'una altra.

Aquí cal fer un parell d'observacions. La primera, que ja es va comentar en l'apartat on s'explica el complement indirecte, és que quan l'únic que

diferenciï un complement de règim verbal i un complement indirecte sigui el significat, aquest es pronominalitzarà malament. La segona és que aquest patró definit anteriorment realment el segueixen molts tipus de CC, però com que es pronominalitzen quasi igual, ni que el programa es confongui com en el cas anterior, els pronominalitzarà bé quasi segur.

També cal veure que aquetes simples condicions funcionen perquè anteriorment s'ha descartat amb molt deteniment que el complement etiquetat com a **obj**, no fos ni complement directe, complement indirecte, complement predicatiu i atribut. Si no, aquesta funció no funcionaria de cap de les maneres.

Just al començament es defineix la funció de nom **complement_regim_verbal()**, que rep per paràmetres únicament les dependències de nom **dep**. Seguidament es defineix una llista de nom **pr**, que emmagatzema totes les preposicions que poden introduir un complement de règim verbal.

Després, s'itera per les dependències i, en cas de trobar que alguna d'aquestes estigui inclosa en la llista **pr**, es comprovarà si es tracta de la preposició *de* o d'una altra. En el cas que la preposició sigui *de*, es retornarà el pronom **en** i, en el cas que no, es retornarà el pronom **hi**. Aquestes instruccions es programen de la següent manera:

```
def complement_regim_verbal( dep):

    pr = ["a", "de", "en", "amb", "per"]

    for e in dep:
        if str(e) in pr:
            if oracio_amb_de(dep): return ["crv", "en"]
            else: return ["crv", "hi"]

    return []
```

4.3.6 Complement circumstancial

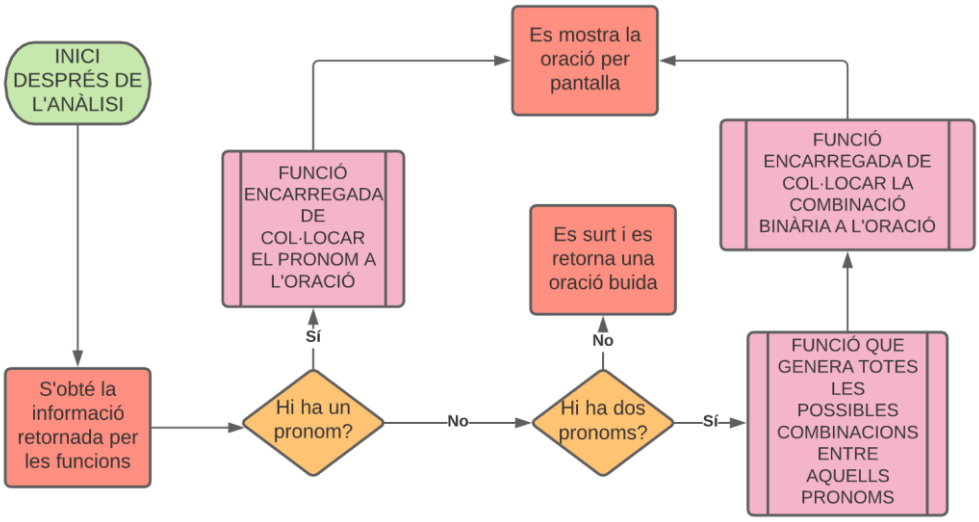
Aquest tipus de complement té moltes possibilitats morfològiques, ja que el poden crear un sintagma nominal, un pronom (feble o fort), i un adverbi o una locució adverbial. Si s'afegeix la dificultat que l'Spacy no etiqueta bé els complements i aquest no només el detecta amb l'etiqueta **obj**, sinó que hi ha

moltes altres possibilitats, s'arriba a la conclusió que amb el model actual de l'Spacy és molt difícil poder pronominalitzar aquests complements correctament.

És per això que, després de moltes hores invertides sense èxit, he decidit que, perquè aquest complement funcioni parcialment, se li atribuirà el tipus de complement circumstancial i es pronominalitzarà per **hi** en els casos que s'hagin descartat totes les opcions possibles d'un complement etiquetat com a **obj**. Tot i que no és la millor solució, ja que hi ha molts casos que no s'estan contemplant, espero en un futur poder solucionar aquest problema.

4.4 Inserció de pronoms i combinacions binàries

Una vegada ja s'han detectat i pronominalitzat tots els complements, l'únic que s'ha de fer és introduir-lo a l'oració, en el cas que només n'hi hagi un; o en el cas que n'hi hagi dos, determinar la combinació binària correcta i després introduir-la en l'oració correctament. Aquest procés s'il·lustra en el següent diagrama de flux:



Il·lustració 22. Diagrama de flux col·locar pronoms

4.4.1 Inserció de pronoms

La funció que col·loca el pronom a l'oració serveix tant per col·locar un sol pronom com per introduir les combinacions binàries a l'oració. És per això que aquesta funció rebrà moltes variables per paràmetres.

Just a l'inici, es defineix la funció de nom **pron_frase()**, que rep per paràmetres el pronom que s'ha de pronominalitzar emmagatzemat en una llista que conté el tipus i les dependències de nom **pron**. També es rep l'oració *"tokenitzada"* amb el nom de **tkora**, una variable booleana de nom **binari** que és certa en el cas que s'estigui pronominalitzant una combinació binària; seguida d'un altra variable booleana de nom **binCdIndet**, que és certa si s'està pronominalitzant un complement directe indeterminat en la combinació binària. Per últim, es rep una variable de tipus cadena de nom **nucliComplement**, que guarda el nucli del complement que s'està pronominalitzant.

Seguidament, es defineix una llista que conté totes les etiquetes amb les quals un complement verbal pot estar etiquetat. Aquesta informació serà útil per determinar si cal incloure certes paraules en l'oració final.

Després es processen les dades rebudes per paràmetres, concretament la llista que conté la variable **pron**. En les variables **pro**, **dep** i **tip**, que representen el pronom, les dependències i el tipus del complement, respectivament, es guarda la informació corresponent.

A continuació, en el cas que no s'estigui col·locant una combinació binària, es mira si s'ha d'apostrofar el pronom i, en cas afirmatiu, s'apostrofa mitjançant la funció externa **apostrofar_article()**. També es guarden totes les dependències del verb, ja que podrien no coincidir amb les dels complements, en una llista de nom **rDep**, mitjançant una funció externa de nom **dependències_verb()**. L'altre estat que es comprova és si el verb de l'oració conté un verb auxiliar, i en cas afirmatiu, es guarda el valor cert en la variable **aux**.

L'última dada que s'ha de processar és guardar la posició que ocupen les dependències en l'oració inicial. D'aquesta manera m'asseguro que no s'inclourà cap dependència que no toqui i que no es traurà cap paraula que no formi part de cap dependència.

El codi que executa totes les instruccions anteriors és el següent:

```
def pron_frase(pron, tkora, binari, binCdIndet, nucliComplement):
    #['cp', 'hi', [], [nucli]]
    comp = ["obj", "ROOT", "advmod", "obl",
    "NMOD", "aux", "amod", 'appos']

    pro, dep, tip = pron[0][1], pron[0][2], pron[0][0]

    if binari == False:
        pro = apostrofar_article(pro, tkora, tip)

    rDep = dependencies_verb(tkora)

    aux = False
    if verb_auxiliar(tkora): aux = True

    depI = []
    for i, e in enumerate(dep):
        depI.append(e.i)
        dep[i] = str(e)
```

Una vegada ja s'han obtingut totes les dades descrites en l'apartat anterior, l'únic que cal fer és anar iterant per l'oració inicial i fer les modificacions pertinents. Hi haurà una variable de nom **s**, que a cada iteració se li afegirà una nova paraula. En algun moment, se li afegirà el pronom que pronominalitza el complement i, d'aquesta manera, quan s'acabi d'iterar, s'obtindrà l'oració pronominalitzada.

A cada iteració es comproven una sèrie de condicions. Primer de tot, es comprova si la paraula en la que s'està iterant és l'auxiliar del verb de l'oració. En el cas que aquesta condició sigui certa, el que es farà serà introduir el pronom i la paraula en la que s'itera a la variable **s**. La segona condició és comprovar si la paraula és el verb de l'oració. En aquest cas, es fan algunes comprovacions extres per si és un verb copulatiu. Recordeu que l'Spacy analitza diferent els verbs copulatius dels predicatius i, fins i tot pot analitzar els verbs copulatius de manera diferent entre ells.

Una vegada ja s'ha comprovat que la paraula no sigui un verb, es comprova que aquella paraula no formi part de cap complement que s'està pronominalitzant, ja que si s'inclogués, el programa no compliria la seva tasca. En aquests condicionals hi ha alguna comprovació extra, ja que en el

cas que s'estigués pronominalitzant un pronom, però es volgués mantenir un complement a l'oració que es pronominalitza per un altre, s'ha de fer algun petit canvi al codi.

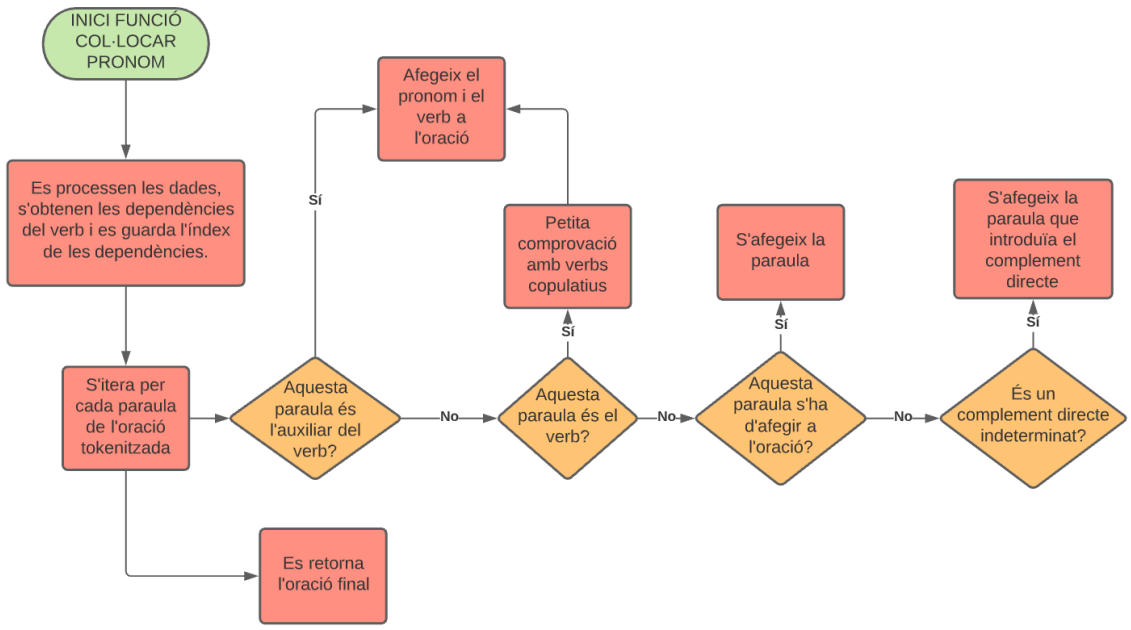
Just al final, només s'ha de comprovar que si s'està pronominalitzant un complement directe indeterminat, s'ha d'afegir la paraula que introdueix el complement darrere del verb.

D'aquesta manera, ja es tindria l'oració amb el pronom col·locat i teòricament sense cap paraula extra. En aquest punt, es retornaria la variable **s** i el programa arribaria al seu final.

El codi que representa aquest procés és el següent:

```
    for i, token in enumerate(tkora):
        if aux and str(token.dep_) == "aux":
            s += pro + " " + str(token) + " " + str(tkora[i+1]) + " "
        elif str(token.dep_) == 'ROOT' and aux == False and tip != 'atr' and tip != 
'atr1':
            s += pro + " " + str(token) + " "
        elif tip == 'atr' and str(token.dep_) == 'cop':
            s += pro + " " + str(token) + " "
        elif tip == 'atr1' and str(token.dep_) == 'ROOT':
            s += pro + " " + str(token.dep_)
        else:
            if token.i not in depI and str(token.dep_) not in comp:
                s += str(token) + " "

            elif (str(token.dep_) in comp) and (token.i not in posNuclis) and 
(token.i not in depI) and (str(token.dep_) != "ROOT") and tip != "atr" and tip != 
"atr1":
                s += str(token) + " "

        if str(token.dep_) == 'ROOT' and (tip == 'cdIndet'or binCdIndet == True) and 
len(dep) != 0 and per == False: s += dep[0] + " "
        elif len(rDep) > 0 and str(token) == str(rDep[-1]) and per == True and 
len(dep) != 0 and (tip == 'cdIndet' or binCdIndet == True): s += dep[0] + " "

    return s
```

El diagrama de flux que realitza tot el procés descrit en aquest apartat és el següent:



Il·lustració 23. Diagrama de flux col·locar pronoms

4.4.2 Combinacions binàries

La funció que determina la combinació binària adequada potser és una mica complicada d'entendre, però realment hi ha una lògica força simple darrere del codi. Per poder fer aquesta tasca, vaig tenir diverses idees sobre com poder dur-la a terme, i al final vaig arribar a la següent solució.

Tal i com es veu a la il·lustració 1, combinacions binàries, es poden col·locar totes les formes possibles dels pronoms febles en una matriu. A cada cel·la hi ha totes les possibles formes dels pronoms, tant al davant com al darrere del verb. Llavors, tenint aquesta taula, el procediment és força senzill. El que s'ha de fer és comprovar els pronoms que hi ha a les fileres o a les columnes i mirar la coordenada que senyalen. Per clarificar una mica aquesta explicació, vegem un exemple amb la següent taula reduïda.

Primer pronom					
	hi	**en**	**ho**	**les**	**la**
es	s'hi	se'n se n'	s'ho	se les	se la se l' / se la
	-s'hi	-se'n	-s'ho	-se-les	-se-la
et	t'hi	te'n te n'	t'ho	te les	te la te l' / te la
	-t'hi	-te'n	-t'ho	-te-les	-te-la
us	us hi	us en us n'	us ho	us les	us la us l' / us la
	-vos-hi -us-hi	-vos-en -us-en	-vos-ho -us-ho	-vos-les -us-les	-vos-la -us-la
em	m'hi	me'n me n'	m'ho	me les	me la me l' / me la
	-m'hi	-me'n	-m'ho	-me-les	-me-la
ens	ens hi	ens en ens n'	ens ho	ens les	ens la ens l' / ens la
	-nos-hi 'ns-hi	-nos-en 'ns-en	-nos-ho 'ns-ho	-nos-les 'ns-les	-nos-la 'ns-la

Il·lustració 24. Taula de combinacions binàries reduïda

Si per exemple, el nostre codi hagués detectat que els pronoms que s'havien de pronominalitzar eren **em** i **les**, s'hauria mirat les posicions que ocupaven els pronoms tant en les fileres com en les columnes. En aquest cas, s'hauria vist que el pronom **les** només estava disponible a la columna 4, i que el pronom **em** a la filera 4. Llavors es determinaria la combinació binària anant a la coordenada (4, 4) de la matriu. Seguidament, es guardaria en una llista les possibles combinacions d'aquells pronoms.

Primer pronom					
	hi	**en**	**ho**	**les**	**la**
es	s'hi	se'n se n'	s'ho	se les	se la se l' / se la
	-s'hi	-se'n	-s'ho	-se-les	-se-la
et	t'hi	te'n te n'	t'ho	te les	te la te l' / te la
	-t'hi	-te'n	-t'ho	-te-les	-te-la
us	us hi	us en us n'	us ho	us les	us la us l' / us la
	-vos-hi	-vos-en	-vos-ho	-vos-les	-vos-la
em	m'hi	me'n me n'	m'ho	me les	me la me l' / me la
	-m'hi	-me'n	-m'ho	-me-les	-me-la
ens	ens hi	ens en ens n'	ens ho	ens les	ens la ens l' / ens la
	-nos-hi 'ns-hi	-nos-en 'ns-en	-nos-ho 'ns-ho	-nos-les 'ns-les	-nos-la 'ns-la

Il·lustració 25. Taula de combinacions binàries reduïda analitzada

Al començament es defineix la funció de nom **bin_pron_frase()**, que rep per paràmetres una llista que emmagatzema tots els pronoms de nom **pron**, i l'oració "tokenitzada" amb el nom de **tkora**.

La llista de nom **pron** estaria formada per dues llistes, que representarien els pronoms d'aquesta manera: **[tipus de complement, pronom, [dependències], nucli del complement]**.

Després es defineixen les variables **m** i **n**, que emmagatzemen els pronoms corresponents a les fileres en el cas de la variable **m**, i les columnes en el cas de la variable **n**, en forma de llista. Aquestes llistes seran clau a l'hora de determinar la combinació binària bona.

A continuació, en el cas del pronom *els*, s'afegeix si és complement directe o indirecte. Això es fa per facilitar la cerca la taula, ja que, tot i que els pronoms siguin iguals, es pronominalitzen de manera diferent si són complement indirecte o complement directe. D'aquest procés no s'ensenyarà el codi, ja que és una mica llarg i no és gaire rellevant en temes de programació. L'únic que cal saber és que un pronom que abans era *els*, s'ha transformat amb *els (CD)*, per exemple.

Seguidament, es posen tots els pronoms amb la seva forma completa per poder fer les cerques a la taula. Aquest procés s'ha de fer perquè en el cas que hi hagués el pronom *m'* a l'oració i s'hagi detectat, s'ha de transformar a *em.* Per determinar això, s'itera pel diccionari que emmagatzema totes les formes dels pronoms i, en el cas que un pronom estigui inclòs en la llista de formes possibles d'aquell pronom, es canviarà el seu valor automàticament. També es comprova que aquell pronom no sigui *els* i se li hagi canviat la forma anteriorment.

Això es programa així:

```
def bin_pron_frase(pron, tkora):

    m = ["es", "et", "us", "em", "ens", "li",
    "els (ci)", "el", "els (cd)", "la", "les", "en"]
    n = ["hi", "en", "ho", "les", "la", "els (cd)",
    "el", "els (ci)", "li", "ens", "em", "us", "et"]


    pr1, pr2 = pron[0], pron[1]

    #
    # CODI QUE AJUSTA ELS PRONOMS "ELS"
    #

    l = ["els (cd)", "els (ci)"]

    for e in PRONOMS.keys():
        if pr1[1] in PRONOMS[e] and pr1[0] not in l:
            pr1[1] = e
        if pr2[1] in PRONOMS[e] and pr1[0] not in l:
```

A continuació, s'ha de determinar la coordenada, si és que existeix, de la combinació binària corresponent. Per fer això, s'accedeix a les variables **n** i

m, i s'emmagatzemen les coordenades. Es mira l'índex que presenta el primer pronom a la llista **n** i **m**, i es fa el mateix procés en l'altre. Llavors, cada coordenada té l'índex d'un pronom a la llista **n**, i l'índex de l'altre a la llista **m**.

En el cas que aquestes combinacions existeixin, i els valors de la coordenada existeixin a dins de la matriu, s'emmagatzemarà en una variable de nom **p** el valor que té la matriu en aquella coordenada. Amb aquest sistema és impossible obtenir duplicats i que el sistema esculli la combinació errònia.

La variable **p** estarà representada per dues llistes. La primera tindrà les possibles combinacions davant del verb, i la segona les possibilitats al darrere del verb.

Això es programaria de la següent manera:

```
    p = []
    m1, m2, n1, n2 = -1, -1, -1, -1

    if pr1[1] in m: m1 = m.index(pr1[1])
    if pr2[1] in m: m2 = m.index(pr2[1])

    if pr1[1] in n: n1 = n.index(pr1[1])
    if pr2[1] in n: n2 = n.index(pr2[1])

    if n2 < len(PRON_BINARIES[m1]) and n2 != -1 and m1 != -1:
        p = PRON_BINARIES[m1][n2]
    elif n1 < len(PRON_BINARIES[m2]) and n1 != -1 and m2 != -1:
        p = PRON_BINARIES[m2][n1]
```

Seguidament, es defineixen tres variables que seran molt importants per saber exactament com i amb quina forma s'han de passar per paràmetres les variables necessàries a la funció **pron_frase()**. Aquestes són una llista de nom **comen,** que guarda totes les possibilitats que presenta una paraula a principi de mot quan s'ha d'apostrofar. No s'està contemplant l'excepció de no apostrofar l'article *la*, que es presenta quan la paraula és femenina i la primera vocal contingui una *hi, hu, i, u* àtones. Seguidament es defineix la variable que emmagatzemarà les dependències dels dos complements de nom **depT**. També es definirà la llista de nom **pT**, que guarda el pronom total que es passa per paràmetres a la funció **pron_frase()**.

A continuació, s'iteraria per cada paraula que conté l'oració "tokenitzada" i, en el cas que aquella paraula fos el nucli del predicat, es començarien a fer les comprovacions pertinents.

La primera comprovació seria mirar si s'ha generat una combinació binària que existeix. En el cas que no existeixi, vol dir que aquells dos pronoms no es poden pronominalitzar alhora. Això, per exemple, passaria en l'oració *Rega flors del jardí.* Si no existeix, el que es fa és concatenar les dues oracions amb la seva pronominalització individual mitjançant la funció **pron_frase()**.

La segona consisteix a comprovar si la combinació binària que precedeix al verb s'hauria d'apostrofar. En cas afirmatiu, es comprova si aquella combinació binària es pot apostrofar i, si es pot, es guarda el que retorna la funció **pron_frase()** amb el pronom apostrofat. En el cas que no es pugui, passarà exactament el mateix, però els paràmetres que se li passaran a la funció seran diferents.

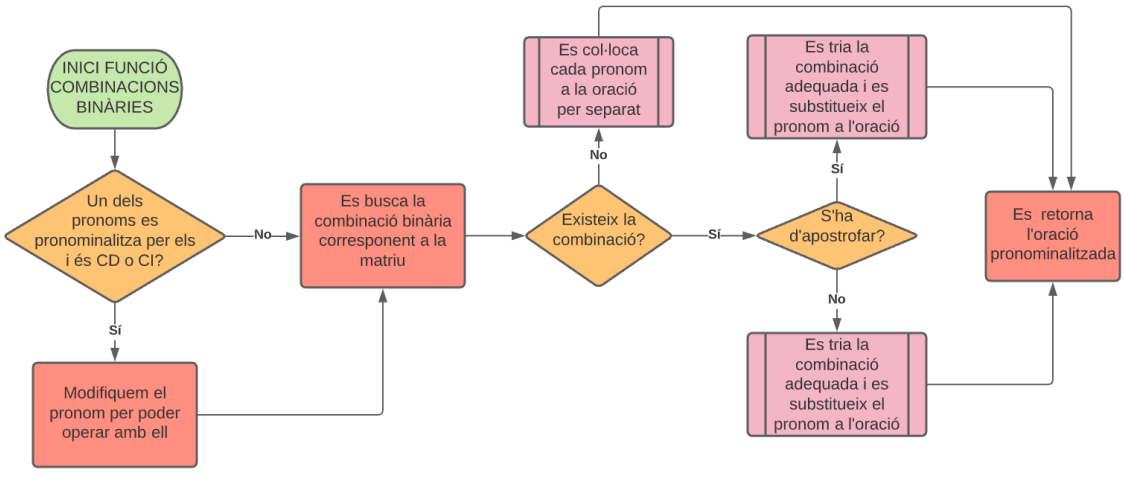
Per acabar, en el cas que aquell pronom no calgui que vagi apostrofat, es passa per paràmetres de la funció **pron_frase()** la combinació binària.

L'oració resultant es retorna a la funció que en un primer lloc ha cridat a la funció **bin_pron_frase()**.

Aquest procés traduït a codi és el següent:

```
    comen = ['a', 'e', 'i', 'o', 'u','ha', 'he',
    'hi', 'ho', 'hu', 'é', 'à', 'ú', 'ò', 'í', 'ó']
    depT = pr1[2] + pr2[2]
    pT = ["bin", " ", depT]

    for token in tkora:
        if str(token.dep_) == 'aux' or str(token.dep_) == 'ROOT':

            if p == []:
                return pron_frase([pron[0]], str(tkora), tkora, False, indet, [nucli1]) + "/ " +
pron_frase([pron[1]], str(tkora), tkora, False, indet, [nucli2])

            s = str(token)

            if s[0] in comen or s[:2] in l:
                if len(p[0]) > 1:
                    pT[1] = [p[0][-1]]
                    return pron_frase([pT], str(tkora), tkora, True, indet, [nucli1, nucli2])
                else:
                    pT[1] = p[0]
                    return pron_frase([pT], str(tkora), tkora, True, indet, [nucli1, nucli2])
            else:
                pT[1] = p[0]
                return pron_frase([pT], str(tkora), tkora, True, indet, [nucli1, nucli2])
```

Aquest procés en un diagrama de flux es veuria de la següent manera:



Il·lustració 26. Diagrama de flux combinacions binàries

SEGONA FASE

5. Creació i publicació del lloc web

En aquest apartat s'explicarà d'una manera molt reduïda i simplificada la creació i publicació del lloc web.

Tot i que aquest no era el meu primer contacte amb el desenvolupament web, al voler integrar un *software* que pronominalitzés en un entorn que fos accessible per tothom, el procés de publicació ha sigut més complicat del que m'hauria imaginat.

5.1 Propòsit de la pàgina web

Tal i com es va comentar a la introducció, un dels meus objectius era visualitzar al màxim tota la recerca feta i explicada anteriorment. Com que quan la vaig acabar, vaig veure que, tot i que el programa tenia certes mancances, funcionava en un gran ventall de casos, vaig decidir que desenvoluparia una pàgina web on es poguessin pronominalitzar oracions simples i s'ensenyés l'anàlisi morfosintàctica que l'Spacy en feia.

5.2 Parts de la pàgina web

Majoritàriament les pàgines web tenen dues parts. Aquestes s'anomenen *front-end* i *back-end.* A continuació s'explicaran aquestes parts amb una mica de detall.

5.2.1 Front-end

Aquesta part s'encarrega de mostrar els elements de la pàgina, que quedin amb un estil bonic, que el disseny web sigui adaptatiu, etc. En resum, és el que veu qualsevol usuari quan entra a la pàgina. Les eines necessàries per desenvolupar la part del *Front-end* són **HTML**, **CSS** i **JavaScript**. Com que no dominava en excés el llenguatge de programació JavaScript, vaig optar per utilitzar Python.

L'HTML s'encarrega de renderitzar elements sense cap mena d'estil a la pàgina web. Aquest llenguatge és la base del text i de l'estructura, però una pàgina feta únicament amb HTML es veu molt pobre. En la il·lustració següent

es veu la pàgina que vaig desenvolupar sense cap mena d'estil aplicat. Tal i com es pot veure, és força pobre, però podria ser totalment funcional.

Il·lustració 27. Pàgina web sense CSS

El CSS s'encarrega d'aplicar estils a cada element de la pàgina web. Tot i que és força senzill saber-lo implementar en una pàgina web, hi ha una gran quantitat d'atributs que cal tenir en compte perquè una pàgina web tingui un bon disseny. Com que jo no soc desenvolupador de *software* especialitzat en el *front-end* de pàgines web i tampoc soc dissenyador, vaig optar per utilitzar unes llibreries de codi obert de nom Bootstrap.[16] Aquestes llibreries m'ofereixen una sèrie d'estils creats per defecte que em facilitaven molt la feina, a partir dels quals l'aspecte de la pàgina web va millorar moltíssim. La il·lustració següent té la mateixa base d'HTML que la il·lustració anterior, però se li han aplicat els estils de la llibreria Bootstrap.

Il·lustració 28. Pàgina web amb CSS

[16] Vegeu: https://getbootstrap.com/.

Javascript és popularment conegut com el llenguatge per fer pàgines web i realment proporciona eines molt potents per interactuar amb els elements de la pàgina d'una manera molt sòlida. Ara que el lloc web està acabat, he vist que si hagués utilitzat Javascript en comptes de Python, algunes tasques s'haurien simplificat molt, ja que Python no és un llenguatge dissenyat per interactuar amb pàgines web.

5.2.2 Back-end

La part del *back-end* és la que gestiona la relació de l'usuari amb el servidor. Per programar aquesta part, vaig optar per utilitzar un *framework* de nom Flask, dissenyat per utilitzar amb Python. Aquesta eina serveix per desenvolupar pàgines web senzilles amb força facilitat i és per això que vaig decidir utilitzar-la.

5.3 Publicació del lloc web

Quan ja vaig tenir el lloc web desenvolupat, vaig decidir publicar-lo amb l'eina Heroku,[17] ja que em proporcionava un servei de servidors gratuït i suportava Python i Flask.

El problema que vaig tenir va ser que tenia integrat tot el *software* que pronominalitzava, incloent-hi les llibreries i el model que em va proporcionar el BSC, a la pàgina web. Llavors, tot i que el servei de Heroku és molt bo, no em permetia penjar un programa que ocupava tant espai i requeria tants recursos.

Com que no sabia com seguir, em vaig posar en contacte amb els investigadors del BSC i em van ajudar a implementar el meu *software* a la pàgina web. La solució va ser separar la pàgina web en dues parts. La primera és exactament igual que abans, amb la petita diferència que tot el *software* que pronominalitza està en un altre servidor, tècnicament anomenat API REST, que s'executa quan rep una petició d'un servidor extern i retorna les dades processades. Separar la pàgina web d'aquesta manera té un gran avantatge, i és que, si jo volgués desenvolupar qualsevol eina que fes us del pronominalitzador, com ara aplicacions mòbils, traductors automàtics, etc., ho podria fer amb molta facilitat. També té un desavantatge i és que mantenir

[17] Vegeu: https://www.heroku.com/home.

aquest servidor en funcionament té un cost econòmic. És per això, que la pàgina web només estarà disponible un temps.

Cal dir que tot aquest procés de publicació ha estat molt complex degut a la complexa instal·lació que té l'Spacy en els dispositius, i l'he pogut dur a terme gràcies a l'expertesa dels investigadors que m'han ajudat. D'una altra manera, no hauria pogut publicar la meva recerca.

5.4 Retroacció rebuda

Una vegada feta la publicació del lloc web, vaig fer-li la màxima publicitat possible per ajudar a tothom que volgués aprendre a pronominalitzar correctament, i rebre retroacció per millorar-la. Tothom que s'ha posat en contacte m'ha transmès unes crítiques molt positives i sorpresa davant del gran encert del *software*. Gràcies a la zona que estava habilitada al lloc web i que es pot veure a la il·lustració següent, hi ha hagut persones que han trobat errors nous, els quals jo desconeixia completament.

Il·lustració 29. Espai habilitat per rebre retroacció

És per això que estic molt agraït amb tothom qui ha dedicat el seu temps per millorar una mica l'eina que he creat.

El lloc web el podeu trobar a: http://www.pronominalitza.cat/.

Conclusions

Si reprenc els quatre objectius que m'havia proposat a l'inici de la recerca, puc veure que hi ha punts que he assolit i d'altres que no tant.

Si començo pel primer objectiu que consistia a crear un *software* funcional que pronominalitzés, crec que puc estar força satisfet perquè l'objectiu s'ha complert força bé. Com ja s'ha anat explicant durant tot el treball, m'he trobat

amb certes limitacions, que malauradament no he pogut superar, però en canvi, n'hi ha d'altres que, amb la meva feina i l'ajuda puntual dels investigadors del BSC, he pogut solucionar. Aquest projecte no el dono per acabat ja que, quan vagin sortint versions noves del model que em proporciona la base del meu treball, el *software* serà més precís i s'equivocarà menys. Quan es solucioni el problema de l'etiquetatge erroni, no tinc cap dubte que el percentatge d'encert del meu *software* millorarà significativament, pel fet que tots els problemes que he tingut estan relacionats amb l'etiquetatge. Per tant, els errors que es produeixen en el complement indirecte, el complement circumstancial i el complement de règim verbal, desapareixeran parcialment. També incrementarà el percentatge d'encert al complement circumstancial, que ara està bastant limitat.

Tot i que segurament no he explicat tots els entrebancs que m'he trobat a l'hora de programar el *software*, ja que han sigut molts i en la redacció s'havien de resumir, puc dir que estic molt content de com he après a solucionar els problemes i treballar cada vegada de forma més ordenada. Cal saber que el codi final té més de 800 línies, cosa que m'ha obligat a aprendre a treballar d'una manera organitzada i polida, perquè un codi d'aquesta extensió pogués ser entenedor per qualsevol programador.

El segon objectiu tractava sobre aprendre com funcionava el model creat pels investigadors del BSC amb profunditat. En aquest cas, no puc estar tan satisfet com amb el primer objectiu. El meu plantejament inicial va ser que crearia el *software* relativament ràpid i que, en el cas que el treball em quedés curt, l'amplificaria estudiant i treballant sobre temes d'IA. Com que vaig tenir molts més problemes dels que m'hauria imaginat creant el *software*, que era el meu objectiu principal, no he pogut amplificar el treball a nivell tècnic de com funciona l'IA creada pel BSC, tot i que m'havia documentat sobre xarxes neuronals i el camp que aplica l'IA en l'àmbit de la lingüística computacional.

Respecte el tercer objectiu de dur a terme la recerca d'una manera transparent i accessible, puc dir que l'he aconseguit d'una manera plena. Això m'ho ha facilitat la plataforma GitHub, ja que he pogut publicar l'estat del *software* en cada moment i fer-lo codi obert oficialment. Com a conseqüència que el codi fos públic, m'he esforçat al màxim per escriure el codi de la

manera més entenedora possible, ja que si qualsevol persona vol seguir millorant aquesta idea pugui començar on l'he deixat jo.

Respecte al quart i últim objectiu que tractava de visualitzar la recerca feta mitjançant un lloc web, crec que l'he assolit correctament. Tot i la meva inexperiència en l'àmbit de desenvolupament de pàgines web, crec que he aconseguit publicar una pàgina que compleix el seu objectiu: difondre la meva recerca, la possibilitat d'ajudar a gent que està aprenent el català, poder resoldre dubtes i rebre *feedback* de la meva recerca. Això ha sigut possible, en gran part, gràcies a la gran comunitat global de programadors que existeix, que ajuda a solucionar qualsevol problema que sorgeix i a proporcionar eines que facilitin la feina del programador, com GitHub o Heroku.

Veient amb perspectiva tota la trajectòria que he anat seguint durant el procés de creació en aquesta recerca, puc veure que he après una quantitat de coses que no m'hauria imaginat en un principi. Tot i que pot sorprendre, en l'àmbit que he après menys ha sigut el de desenvolupar el *software* que pronominalitza. Això és degut al fet que partia d'una base consolidada dels coneixements que necessitava per programar un *software* d'aquestes característiques. No obstant això, aquests coneixements m'han permès tenir una fluïdesa a l'hora de programar molt gran, sense la qual aquest treball no hauria pogut arribar tan lluny.

En un àmbit que he après molt ha sigut el de desenvolupar un projecte d'unes característiques grans mitjançant Git. Tot i que pot semblar una minúcia, Git m'ha ajudat moltíssim a mantenir el codi estructurat i m'ha estalviat molt temps a l'hora de buscar versions anteriors del *software*. També m'ha permès fer el codi públic per tots els programadors que hi tinguin interès i, si hi volen col·laborar, que ho puguin fer d'una manera directa.

Aquest trajecte m'ha introduït en el món de la lingüística computacional, desconeguda per mi fins el moment en què el meu tutor me la va introduir. Gràcies a això, he vist que en aquest àmbit encara es pot fer molta feina i, sobretot, millorar i proporcionar materials de lingüística computacional centrats en el català, que són força menors comparats amb altres llengües.

Seguint amb aquesta línia, he descobert el mòdul Spacy i una petita part del potencial que es pot extreure a una eina tant potent.

En un altre sector que he après moltíssim ha sigut el de desenvolupament de llocs web. I és que, tot i que jo hi havia tingut un primer contacte, mai m'havia atrevit plenament a programar i desenvolupar una pàgina web, ja fos perquè s'havia de treballar en molts àmbits diferents i no em veia capaç, o per prioritats a l'hora de triar projectes que em motivessin. Aquesta recerca i el fet de pensar que el *software* que havia generat podia ajudar a la gent m'ha motivat a fer el pas final i, tot i les adversitats, he aconseguit acabar publicant el meu lloc web.

Respecte a tot el treball, he d'agrair immensament l'ajuda i l'atenció que he tingut en tot moment per part dels investigadors del Barcelona Supercomputing Center, ja que sense ells, aquest treball no hauria estat possible. M'han ajudat sempre i quan els he necessitat, tant en l'apartat de desenvolupament del *software*, com en l'apartat de publicar la pàgina web. Han jugat un paper clau perquè el treball pogués arribar a bon port.

Finalment, la gran inversió de temps en aquesta recerca ha reafirmat la meva passió per la programació i espero que serveixi com a base d'un futur, on pugui utilitzar totes les eines tecnològiques apreses per invertir-les en la recerca, ja sigui en la lingüística computacional en català, o en qualsevol altre tema d'intel·ligència artificial.

Bibliografia

Ancora (2021). *Corpus* http://clic.ub.edu/corpus/es/ancora.

Bootstrap (2021). https://getbootstrap.com/.

DIEC (2021). https://dlc.iec.cat/.

Flask (2021). https://flask.palletsprojects.com/en/2.0.x/.

Git (2021). https://git-scm.com/.

GitHub (2021). https://github.com/.

GitHub Pages (2021). https://pages.github.com/.

Heroku (2021). https://www.heroku.com/home.

Institut d'Estudis Catalans (2018). *Gramàtica essencial de la llengua catalana*. https://geiec.iec.cat/gramatica.asp.

Optimot (2021). https://aplicacions.llengua.gencat.cat/llc/AppJava/index.html.

Parlament de Catalunya, Departament d'Assessorament Lingüístic (2021). *Combinacions de dos pronoms febles.* https://www.parlament.cat/document/intrade/240096.

Plataforma Codelearn (2021). https://codelearn.es/.

Spacy (2021). *spaCy 101: Everything you need to know* https://spacy.io/usage/spacy-101.

Universal dependencies (2021) *UD for Catalan*. https://universaldependencies.org/ca/index.html.