U3 :: Procesamiento del Lenguaje Natural

2. Análisis del lenguaje natural con expresiones regulares

2.1 Introducción

Descubrir nuevas palabras "secretas" en documentos desclasificados de la CIA puede parecer una misión propia de un servicio de inteligencia extranjero, o detectar sesgos de género en las novelas de Harry Potter, una tarea para un profesor de literatura. Sin embargo, todo ello es posible mediante ***el análisis sintáctico del lenguaje natural con expresiones regulares***.

Si bien es posible que no pensemos mucho en la estructura de las oraciones mientras escribimos, las elecciones sintácticas que hacemos son fundamentales para garantizar que el texto tenga significado. El análisis de la estructura de la oración, así como la elección de palabras, no solo puede proporcionar información sobre la connotación de un texto, sino que también puede destacar los sesgos de su autor o descubrir conocimientos adicionales que incluso una lectura profunda y rigurosa del texto podría no revelar.

Mediante el uso *El módulo de expresión regular de Python* y la librería NLTK se pueden encontrar palabras clave y descubrir dónde y con qué frecuencia se usan en diferentes partes de un discurso formando patrones mediante los que un texto adquiere verdadero significado.

2.2. Interpretación y emparejamiento

Antes de sumergirnos en un análisis sintáctico de mayor complejidad, comenzaremos con expresiones regulares básicas en Python usando el módulo *re*. Pero previamente cabe aclarar qué es una expresión regular.

Una **expresión regular** es un patrón que describe una cierta cantidad de texto. Por ejemplo, sabes que los correos electrónicos son siempre como:

username@domain.extension

Si queremos describir el patrón de un correo electrónico, diríamos algo como esto: Comenzando con un nombre de usuario (una combinación de letras y números), seguido de un símbolo arroba **@**, seguido del dominio (otra combinación de letras y números) seguido de la extensión (que comienza con un punto . seguido de una combinación de letras solamente).

El proceso de describir el patrón de un correo electrónico es el mismo proceso que seguirá cuando desee crear una expresión regular. Si deseas profundizar en la construcción de expresiones regulares, en estas páginas encontrarás un tutorial útil:

- Tutorial de Regex: Aprender con ejemplos de expresiones regulares
- RegexOne: Learn Regular Expressions with simple, interactive exercises.

Y en esta otra, una forma de facilitar su construcción y revisión:

- RegExr is an online tool to learn, build, & test Regular Expressions

Dicho esto, el primer método de un objeto "re" que explicaremos es ***.compile()***. Este método toma un patrón de expresión regular como argumento e interpreta el patrón en un objeto de expresión regular, que luego se puede usar para buscar texto coincidente. El objeto de expresión regular a continuación coincidirá exactamente con 4 caracteres en mayúscula o minúscula.

```
regular_expression_object = re.compile("[A-Za-z]{4}")
```

Los objetos de expresión regular tienen un método llamado ***.match()*** que toma una cadena de texto como argumento y busca una *única* coincidencia con la expresión regular que *comienza al principio* de la cadena. Para ver si su expresión regular coincide con la cadena "Toto", puedes hacer lo siguiente:

```
result = regular_expression_object.match ("Toto")
```

Si *.match()* encuentra una coincidencia que comienza al principio de la cadena, devolverá una coincidencia objeto. El objeto de coincidencia permite saber con qué fragmento de texto coincide la expresión regular y en qué índice comienza y termina la coincidencia. Si no hay coincidencia, .match() devolverá *None*.

Con el objeto de coincidencia almacenado en el resultado, se puede acceder al texto coincidente llamando a *result.group(o)*. Si se usa una expresión regular que contiene grupos de captura, se puede acceder a estos grupos llamando a *.group()* con el grupo de captura numerado apropiadamente como argumento.

En lugar de interpretar la expresión regular primero y luego buscar una coincidencia en líneas de código separadas, puede simplificar su coincidencia en una línea:

```
result = re.match ("[A-Za-z] {4}", "Toto" )
```

Con esta sintaxis, el método *.match()* toma un patrón de expresión regular como primer argumento y una cadena como segundo argumento.

Ejemplo resumen

```
import re

# characters are defined

character_1 = "Dorothy"

character_2 = "Henry"

# compile your regular expression here

regular_expression = re.compile("[a-zA-Z]{7}")

# check for a match to character_1 here

result_1 = re.match(regular_expression, character_1)

#print(result_1)

# store and print the matched text here

match_1 = result_1.group(0)

print(match_1)

# compile a regular expression to match a 7 character string of word characters and check
for a match to character_2 here

result_2 = re.match("[a-zA-Z]{7}",character_2)

print(result_2)
```

2.3. Búsqueda

Se puede hacer que las coincidencias de expresiones regulares sean aún más dinámicas con la ayuda del método **.search()**. A diferencia de .match(), que solo encontrará coincidencias al comienzo de una cadena, *.search()* buscará de izquierda a derecha en todo el texto y devolverá un objeto de coincidencia para la primera coincidencia con la expresión regular dada. Si no se encuentra ninguna coincidencia, *.search()*

devolverá *None*. Por ejemplo, para buscar una secuencia de caracteres de 8 palabras en la cadena ¿Eres un Munchkin ?:

```
result = re.search("\w{8}","Are you a Munchkin?")
```

Usando *.search()* en la cadena de arriba encontrará una coincidencia de "Munchkin", mientras que usar *.match()* en la misma cadena devolvería *None*!

Hasta ahora, hemos utilizado métodos que solo devuelven un fragmento de texto coincidente. ¿Qué sucede si desea encontrar todas las apariciones de una palabra o palabra clave en un fragmento de texto para determinar un recuento de frecuencia? En ese caso cabe usar **.findall()**

Dada una expresión regular como primer argumento y una cadena como segundo argumento, *.findall()* devolverá una lista de todas *las* coincidencias *no superpuestas* de la expresión regular en la cadena. Considera el siguiente fragmento de texto:

```
text = "Everything is green here, while in the country of the Munchkins blue was the favorite color. But the people do not seem to be as friendly as the Munchkins, and I'm afraid we shall be unable to find a place to pass the night."
```

Para encontrar todas *las secuencias no superpuestas* de caracteres de 8 palabras en la oración, se puede hacer lo siguiente:

```
list_of_matches = re.findall("\w{8}",text)
```

.findall() devolverá la lista ['Everything', 'Munchkin', 'favorito', 'amistoso', 'Munchkin'].

Ejemplo resumen

```
import re

# import L. Frank Baum's The Wonderful Wizard of Oz

oz_text = open("the_wizard_of_oz_text.txt",encoding='utf-8').read().lower()

# search oz_text for an occurrence of 'wizard' here

found_wizard = re.search("wizard", oz_text)
```

```
# find all the occurrences of 'lion' in oz_text here

all_lions = re.findall("lion", oz_text)

# store and print the length of all_lions here

number_lions = len(all_lions)

print(number_lions)
```

2.4. Etiquetado de parte del discurso ("*Part-of-Speech* Tagging")

Si bien es útil hacer coincidir y buscar patrones de caracteres individuales en un texto, a menudo puede resultar de más utilidad que analizar el texto palabra por palabra enfocarse en la parte del discurso de cada palabra. Este proceso de identificación y etiquetado de la parte del discurso de las palabras se conoce como ***etiquetado gramatical***.

La siguientes son algunas de las categorías gramaticales con las que podemos trabajar:

- **Sustantivo:** el nombre de una persona (Ramona, clase), lugar, cosa (libro de texto) o idea (PNL)
- **Pronombre:** una palabra que se usa en lugar de un sustantivo (ella, ella).
- **Determinante:** una palabra que introduce, o "determina ", Un sustantivo (el)
- **Verbo:** expresa acción (estudiar) o ser (son, tiene)
- **Adjetivo:** modifica o describe un sustantivo o pronombre (nuevo)
- **Adverbio:** modifica o describe un verbo, un adjetivo u otro adverbio (felizmente)
- **Preposición:** una palabra colocada antes de un sustantivo o pronombre para formar una frase que modifica otra palabra en la oración (en)
- **Conjunción:** una palabra que une palabras, frases o cláusulas (e)
- **Interjección:** una palabra que se usa para expresar emoción (Wow)

Se puede automatizar el proceso de etiquetado de parte del discurso con la función **pos_tag()** de NLTK. La función toma un argumento, una lista de palabras en el orden en que aparecen en una oración, y devuelve una lista de tuplas, donde la primera entrada en la tupla es una palabra y la segunda es la etiqueta de parte del discurso.

Dada la siguiente lista de palabras en inglés extraídas de una oración anterior:

```
word_sentence = ['do', 'you', 'suppose', 'oz', 'could', 'give', 'me', 'a', 'heart', '?']
```

Se puede etiquetar las partes del discurso de la siguiente manera:

```
part_of_speech_tagged_sentence = pos_tag (word_sentence)
```

La llamada a pos_tag () devolverá lo siguiente:

```
[('do', 'VB'), ('you', 'PRP'), ('suppose', 'VB'), ('oz', 'NNS'), ('could', 'MD'), ('give', 'VB'), ('me', 'PRP'), ('a', 'DT'), ('heart', 'NN'), ('?', '.')]
```

En inglés, aparecen las abreviaturas en lugar de la parte completa del nombre del discurso. Algunas de estas abreviaturas son: **NN** para sustantivos, **VB** para verbos, **RB** para adverbios, **JJ** para adjetivos y **DT** para determinantes. La lista completa de etiquetas de parte del discurso y sus abreviaturas se puede encontrar aquí.

```
import nltk

from nltk import pos_tag

from word_tokenized_oz import word_tokenized_oz

# save and print the sentence stored at index 100 in word_tokenized_oz here

witches_fate = word_tokenized_oz[100]

print(witches_fate)

# create a list to hold part-of-speech tagged sentences here

pos_tagged_oz = list()

# create a for loop through each word tokenized sentence in word_tokenized_oz here

for word_tokenized_sentence in word_tokenized_oz:

 # part-of-speech tag each sentence and append to pos_tagged_oz here

 pos_tagged_oz.append(pos_tag(word_tokenized_sentence))

# store and print the part-of-speech tagged sentence at index 100 in witches_fate_pos here
```

```
witches_fate_pos = pos_tagged_oz[100]

print(witches_fate_pos)
```

2.5. Introducción a *Chunking* ("fragmentación")

Una vez contamos con un texto etiquetado podemos usar expresiones regulares para encontrar patrones en la estructura de la oración que nos den información sobre su significado. Esta técnica de agrupar palabras por su etiqueta de parte del discurso se llama ***fragmentación***.

Mediante la fragmentación se puede definir un patrón gramatical basado en expresiones regulares. Más tarde, se puede encontrar coincidencias que no se superpongan, o *sintagmas*, en las oraciones cuyas partes se han etiquetado como parte del discurso para un texto dado.

La expresión regular que se crea para encontrar sintagmas se llama *fragmentos gramaticales*. Un fragmento gramatical se puede describir de la siguiente manera:

chunk_grammar = **"AN: {<JJ> <NN>}"**

- **AN** es un nombre definido por el usuario para el tipo de fragmento que está buscando. Puedes usar cualquier nombre que tenga sentido dada su gramática. En este caso, AN significa *adjetivo-sustantivo*
- Un par de llaves **{}** rodean la gramática del fragmento real
- **<JJ>** funciona de manera similar a una clase de caracteres regex, coincidiendo con cualquier adjetivo
- **<NN>** coincide con cualquier sustantivo, singular o plural

El fragmento gramatical anterior por lo tanto, empareja cualquier adjetivo seguido de un sustantivo.

Para usar el sintagma definido, se debe crear un objeto NLTK *RegexpParser* y pasarle un *fragmento gramatical* como argumento.

```
chunk_parser = RegexpParser(chunk_grammar)
```

A continuación, puede utilizar el método **.parse()** del objeto *RegexpParser*, que toma una lista de palabras etiquetadas como parte de la oración como argumento e identifica dónde aparecen dichos fragmentos en la oración.

Si tomamos la siguiente oración etiquetada de parte del discurso:

```
pos_tagged_sentence = [('where', 'WRB'), ('is', 'VBZ'), ('the', 'DT'), ('emerald', 'JJ'), ('city', 'NN'), ('?', '.')]
```

Podemos fragmentar la oración para encontrar cualquier adjetivo seguido de un sustantivo con lo siguiente:

```
chunked = chunk_parser.parse (pos_tagged_sentence)
```

Ejemplo

```
from nltk import RegexpParser, Tree
# definimos un fragmento gramatical del tipo 'adjetivo + nombre'
chunk_grammar = "AN:{<JJ><NN>}"
# creamos un objeto 'RegexpParser'
chunk_parser = RegexpParser(chunk_grammar)
# buscamos el patrón en el etiquetado del párrafo de Turing anterior
turing_chunked = chunk_parser.parse(turing_pos)
# imprimimos en forma de árbol mediante el método 'pretty_print' la frase
fragmentada
Tree.fromstring(str(turing_chunked)).pretty_print()
```

2.5.1. Fragmentación de sintagmas nominales

Si bien se puede dividir cualquier secuencia de partes del discurso que se desee, existen ciertos tipos de fragmentos que son útiles desde el punto de vista lingüístico para determinar el significado y el sesgo en un fragmento de texto. Uno de estos tipos de fragmentación es *NP-chunking*, o ***fragmentación de sintagmas nominales***. Un *sintagma nominal* es un sintagma que contiene un sustantivo y opera, como unidad, como sustantivo.

Una forma popular de sintagma nominal comienza con un *determinante* DT, que especifica el sustantivo al que se hace referencia, seguido de cualquier número de *adjetivos* JJ, que describen el sustantivo, y termina con un sustantivo NN.

Tomemos la siguiente oración etiquetada en sus partes del discurso:

[('we', 'PRP'), ('are', 'VBP'), ('so', 'RB'), ('grateful', 'JJ'), ('to', 'TO'), ('you', 'PRP'), ('for', 'IN'), ('having', 'VBG'), ('killed', 'VBN'), ('the', 'DT'), ('wicked', 'JJ'), ('witch', 'NN'), ('of', 'IN'), ('the', 'DT'), ('east', 'NN'), (',', ','), ('and', 'CC'), ('for', 'IN'), ('setting', 'VBG'), ('our', 'PRP$'), ('people', 'NNS'), ('free', 'VBP'), ('from', 'IN'), ('bondage', 'NN'), ('.', '.')]

En ella, se puede encontrar tres frases nominales de la forma descrita anteriormente:

- (('the', 'DT'), ('wicked', 'JJ'), ('witch', 'NN'))
- (('the', 'DT'), ('east', 'NN'))
- (('bondage', 'NN'))

Con la ayuda de una expresión regular, podemos encontrar fácilmente todas los sintagmas nominales no superpuestos en un fragmento de texto. Al igual que en las expresiones regulares normales, se pueden usar cuantificadores para indicar cuántos de cada parte de la oración que se desea hacer coincidir.

La expresión regular correspondiente a un sintagma nominal se puede expresar de la siguiente manera:

chunk_grammar = "NP: {<DT>?<JJ>*<NN>}"

- NP es el nombre definido por el usuario del fragmento que está buscando. En este caso, NP significa sintagma nominal ("normal phrase").
- <DT> coincide con cualquier determinante.
- ? es un *cuantificador opcional*, que coincide con o o 1 determinantes.
- <JJ> coincide con cualquier adjetivo.
- * es el cuantificador de *Kleene* en que coincide con o o más ocurrencias de un adjetivo.
- <NN> coincide con cualquier sustantivo, singular o plural.

La identificación de sintagmas nominales permite:

- realizar un análisis de frecuencias e identificar frases nominales importantes y recurrentes.
- usar estos *NP-chunks* como pseudo-temas y etiquetar artículos y documentos en función del número de sintagmas de este tipo que aparezcan.
- orientar cierto análisis a observar, por ejemplo, las elecciones de adjetivos que hace un autor para diferentes sustantivos. En última instancia, la interpretación del significado y el tipo de sintagma o fragmento NP y de la frecuencia de aparición del mismo dependerá, como puede intuirse, del conocimiento del texto con el que se esté trabajando.

Ejemplo

```
import nltk
from nltk import RegexpParser, Tree, pos_tag
from nltk.tokenize import word_tokenize
from nltk.tokenize import sent_tokenize
#nltk.download('punkt')
#nltk.download('averaged_perceptron_tagger')

# definimos un fragmento gramatical del tipo sintagma nominal
chunk_grammar = "NP:{<DT>?<JJ>*<NN>}"
```

```
# creamos un objeto 'RegexpParser'
chunk_parser = RegexpParser(chunk_grammar)

turing_text = "Born in Maida Vale, London, Turing was raised in southern
England. He graduated at King's College, Cambridge, with a degree in
mathematics. Whilst he was a fellow at Cambridge, he published a proof
demonstrating that some purely mathematical yes-no questions can never be
answered by computation and defined a Turing machine, and went on to prove the
halting problem for Turing machines is undecidable. In 1938, he obtained his PhD
from the Department of Mathematics at Princeton University."

# Tokenizamos el párrafo anterior
tokenized_sentences = list()
for sentence in sent_tokenize(turing_text):
 tokenized_sentences.append(word_tokenize(sentence))
 print(sentence)

# Ahora lo fragmentamos previo etiquetado de sus palabras
turing_chunked = list()
for words in tokenized_sentences:
 turing_chunked.append(chunk_parser.parse(pos_tag(words)))

# Imprimimos cada frase en forma de árbol
for number_sentence in range(len(turing_chunked)):
 Tree.fromstring(str(turing_chunked[number_sentence])).pretty_print()
```

2.5.2. Fragmentación de frases verbales

Otro tipo de fragmentación es la fragmentación de *VP*, o ***fragmentación de sintgamas verbales***. Una *frase verbal* es una frase que contiene un verbo y sus complementos, objetos o modificadores.

Las frases verbales pueden tener una variedad de estructuras, y aquí se considerarán dos. La primera estructura comienza con un verbo VB de cualquier tiempo, seguido de un sintagma nominal, y termina con un adverbio opcional RB de cualquier forma. La segunda estructura cambia el orden del verbo y el sintagma nominal, pero también termina con un adverbio opcional.

Ambas estructuras se consideran porque las frases verbales de cada forma tienen esencialmente el mismo significado. Por ejemplo, considere las frases verbales etiquetadas de parte del discurso que se dan a continuación:

- (('said', 'VBD'), ('the', 'DT'), ('cowardly', 'JJ'), ('lion', 'NN'))
- ('the', 'DT'), ('cowardly', 'JJ'), ('lion', 'NN')), (('said', 'VBD'),

El fragmento de gramática para encontrar la primera forma de la frase verbal se da a continuación:

chunk_grammar = "VP: {<VB. *> <DT>? <JJ> * <NN> <RB.?>?}"

- **VP** es el identificador definido por el usuario oara el fragmento que está buscando. En este caso, VP significa "verbal phrase".
- **<VB. *>** coincide con cualquier verbo usando .como comodín y el cuantificador * para coincidir con o o más apariciones de cualquier carácter. Esto asegura verbos coincidentes de cualquier tiempo (por ejemplo, VB para tiempo presente, VBD para tiempo pasado o VBN para participio pasado)
- **<DT>? <JJ> * <NN>** coincide con cualquier sintagma nominal
- **<RB.?>** coincide con cualquier adverbio que use .como comodín y el *cuantificador opcional* para que coincida con o o 1 aparición de cualquier carácter. Esto asegura que coincida con cualquier forma de adverbio (RB regular, RBR comparativo o RBS superlativo)
- ? es un *cuantificador opcional*, que coincide con o o 1 adverbios

El chunk gramática para la segunda forma de frase verbal se da a continuación:

chunk_grammar = "VP: {<DT>? <JJ>*<NN><VB.*> <RB.?>?}"

Al igual que con *NP-chunks* podemos encontrar todos los sintagmas verbales en un texto y realizar un análisis de frecuencia para identificar frases verbales importantes y recurrentes. Estas construcciones pueden dar una idea de qué tipo de acción realizan los diferentes personajes o cómo el autor describe las acciones que estos realizan.

Una vez más, esta es la parte del análisis en la que uno puede ser creativo y usar su propio conocimiento sobre el texto con el que estás trabajando para encontrar ideas interesantes.

Ejemplo

```
import nltk
from nltk import RegexpParser, Tree, pos_tag
from nltk.tokenize import word_tokenize
from nltk.tokenize import sent_tokenize
#nltk.download('punkt')
#nltk.download('averaged_perceptron_tagger')
# definimos un fragmento gramatical del tipo sintagma nominal
chunk_grammar = "VP: {<VB. *> <DT>? <JJ> * <NN> <RB.?>?}"
# creamos un objeto 'RegexpParser'
chunk_parser = RegexpParser(chunk_grammar)

turing_text = "..."
```

```
# Tokenizamos el párrafo anterior
tokenized_sentences = list()
print("SENTENCES: \n")
for sentence in sent_tokenize(turing_text):
 tokenized_sentences.append(word_tokenize(sentence))
 print(sentence)

# Ahora lo fragmentamos previo etiquetado de sus palabras
turing_chunked = list()
for words in tokenized_sentences:
 turing_chunked.append(chunk_parser.parse(pos_tag(words)))

# Imprimimos cada frase en forma de árbol
print("\nREPRESENTATION: ")
for number_sentence in range(len(turing_chunked)):
 Tree.fromstring(str(turing_chunked[number_sentence])).pretty_print()
```

2.5.3. Filtrado de fragmentos

Otra opción de la que disponemos para buscar fragmentos en un texto es el *filtrado de fragmentos*. ***El filtrado de fragmentos*** permite definir qué partes del discurso no se desea y eliminarlas.

Un método extendido para realizar el filtrado de fragmentos es agrupar una oración completa y luego indicar qué partes del discurso se deben filtrar. Si las partes filtradas del discurso están en medio de un fragmento, se dividirá el fragmento en dos fragmentos separados. La siguiente sería una expresión regular que permitiría realizar el filtrado de fragmentos:

```
chunk_grammar = """NP: {<.*>+}
                          }<VB.?|IN>+{"""
```

- NP es el nombre definido por el usuario del fragmento que está buscando. En este caso, NP significa sintagma nominal
- Los corchetes {} indican qué partes del discurso está fragmentando. < . *> + coincide con cada parte de la oración en la oraciónla oración
- Los corchetes invertidos }{ indican qué partes deseas filtrar del fragmento. <VB.?|IN>+ filtrará cualquier verbo o preposición

El filtrado de fragmentos proporciona una alternativa una forma de buscar en un texto y encontrar los fragmentos de información útiles para tu análisis.

Ejemplo

```
# EJEMPLO DE FILTRADO DE FRAGMENTACIÓN

import nltk
from nltk import RegexpParser, Tree, pos_tag
from nltk.tokenize import word_tokenize
from nltk.tokenize import sent_tokenize
#nltk.download('punkt')
#nltk.download('averaged_perceptron_tagger')
# definimos un fragmento gramatical general
grammar = "Chunk: {<.*>+}"
# creamos un objeto 'RegexpParser'
parser = RegexpParser(grammar)

turing_text = "Born in Maida Vale, London, Turing was raised in southern
England. He graduated at King's College, Cambridge, with a degree in
mathematics. Whilst he was a fellow at Cambridge, he published a proof
demonstrating that some purely mathematical yes-no questions can never be
answered by computation and defined a Turing machine, and went on to prove the
halting problem for Turing machines is undecidable. In 1938, he obtained his PhD
from the Department of Mathematics at Princeton University."

# Tokenizamos el párrafo anterior
tokenized_sentences = list()
for sentence in sent_tokenize(turing_text):
 tokenized_sentences.append(word_tokenize(sentence))

# Ahora lo fragmentamos previo etiquetado de sus palabras
turing_chunked = list()
for words in tokenized_sentences:
 turing_chunked.append(parser.parse(pos_tag(words)))

# Imprimimos cada frase en forma de árbol
print("\nREPRESENTATION: ")
for number_sentence in range(len(turing_chunked)):
 Tree.fromstring(str(turing_chunked[number_sentence])).pretty_print()

# recuperamos el fragmento gramatical general al que indicamos las partes a
filtrar
chunk_grammar = """NP: {<.*>+}
                     }<VB.?|IN>+{"""
# creamos un objeto 'RegexpParser'
```

```
chunk_parser = RegexpParser(chunk_grammar)

turing_text = "Born in Maida Vale, London, Turing was raised in southern
England. He graduated at King's College, Cambridge, with a degree in
mathematics. Whilst he was a fellow at Cambridge, he published a proof
demonstrating that some purely mathematical yes-no questions can never be
answered by computation and defined a Turing machine, and went on to prove the
halting problem for Turing machines is undecidable. In 1938, he obtained his PhD
from the Department of Mathematics at Princeton University."

# Tokenizamos el párrafo anterior
tokenized_sentences = list()
for sentence in sent_tokenize(turing_text):
 tokenized_sentences.append(word_tokenize(sentence))

# Ahora lo fragmentamos previo etiquetado de sus palabras
turing_chunked = list()
for words in tokenized_sentences:
 turing_chunked.append(chunk_parser.parse(pos_tag(words)))

# Imprimimos cada frase en forma de árbol
print("\nREPRESENTATION WITHOUT VP: ")
for number_sentence in range(len(turing_chunked)):
 Tree.fromstring(str(turing_chunked[number_sentence])).pretty_print()
```

Revisión

Con lo anterior tenemos un conjunto de herramientas para profundizar en cualquier parte de los datos de texto y realizar análisis en lenguaje natural con expresiones regulares. ¿Qué sesgo se puede descubrir?

Repasemos lo que ha aprendido:

- Los métodos *.compile()* y *.match()* del módulo *re* permiten ingresar cualquier patrón de expresiones regulares y buscar una sola coincidencia al comienzo de una pieza de texto.
- El método .search() del módulo *re* permite encontrar una única coincidencia con un patrón de expresiones regulares en cualquier lugar de una cadena, mientras que el método *.findall()* encuentra todas las coincidencias de un patrón de expresiones regulares en una cadena.
- El etiquetado de parte del discurso identifica y etiqueta la parte del discurso de las palabras en una oración y se puede realizar en NLTK usando la función *pos_tag().*
- *La fragmentación* agrupa los patrones de palabras por su etiqueta de parte del discurso. La fragmentación se puede realizar en NLTK definiendo un fragmento de gramática de fragmentos utilizando la sintaxis de expresión regular y llamando al método .parse() de *RegexpParser* en una oración tokenizada por palabra.
- *NP-chunking* junta un determinante opcional DT, con cualquier número de adjetivos JJ y un sustantivo NN para formar un sintagma nominal. La frecuencia de diferentes *NP-chunks* puede dar una idea de los temas importantes en un texto o demostrar cómo un autor aborda diferentes temas.
- *VP-chunking* agrupa un verbo VB, un sintagma nominal y un adverbio opcional RB para formar un sintagma verbal. La frecuencia de diferentes VP-chunks puede dar una idea de qué tipo de acción toman los diferentes sujetos o cómo un autor describe las acciones que toman los diferentes sujetos, lo que podría indicar un sesgo.
- *El filtrado* de fragmentos proporciona un medio alternativo de fragmentación al especificar qué partes de la oración no desea en un fragmento y eliminarlas.

2.6. Fuentes complementarias

- **Extracción de información del texto**
  - https://www.NLTK.org/book/ch07.html
- **Chunking in NLP: decoded**
  - https://towardsdatascience.com/chunking-in-nlp-decoded-b4a71b2b4e24
- **NLTK: Etiquetado morfológico (part-of-speech tagging)**
  - https://colab.research.google.com/github/vitojph/kschool-nlp-18/blob/master/notebooks/nltk-pos.ipynb#scrollTo=Djf_MmiHbOSZ