



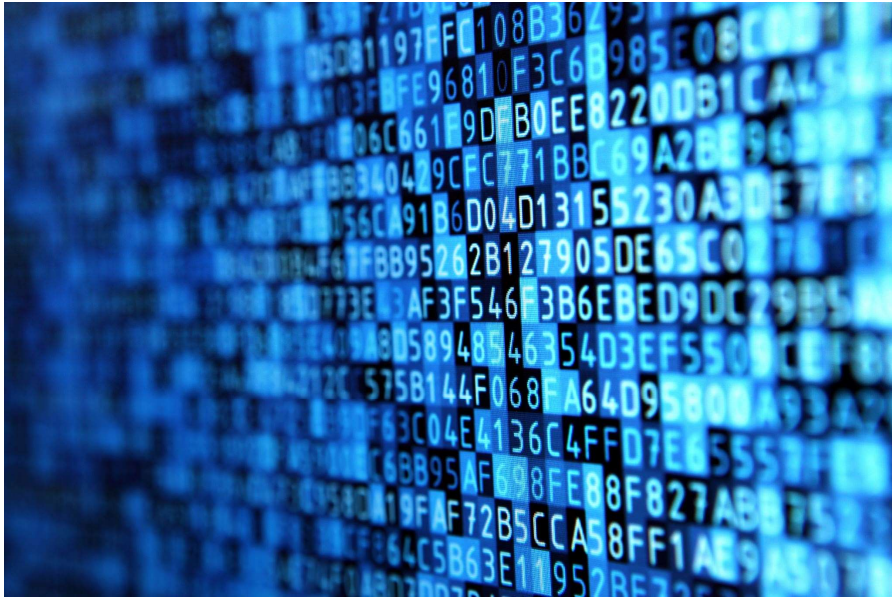
Maestría en Inteligencia Artificial Aplicada (MNA)

# Pre-procesamiento y más tipos de datos

Procesamiento de Lenguaje Natural (NLP)

Luis Eduardo Falcón Morales

# Continuando con etapa de Tokenización, regex y tipos de datos



En esta semana seguiremos estudiando otros tipos de datos. En particular, veremos los diccionarios, los cuales serán un tipo de dato muy importante durante la etapa de pre-procesamiento y tokenización de documentos de texto.

Por otro lado, la etapa de tokenización nos lleva a generar las unidades mínimas de información y a partir de las cuales definiremos los diccionarios.

Todo lo anterior se sigue combinando con el tema de expresiones regulares (regex).

## Tipos de datos

Continuemos con el estudio de los tipos de datos que usualmente manejamos en un problema de análisis de textos:

Estudiados la  
semana pasada

- Numeric & Boolean : numérico y booleano
- Strings : cadena de caracteres
- Lists : listas
- Dictionaries : diccionarios
- Sets : conjuntos
- Tuples : tuplas
- Series : series

## Diccionarios

Un diccionario nos permite guardar información en formato **clave:valor**, **key:value**.

Existe una gran variedad de aplicaciones para este tipo de datos: directorios telefónicos; lista de estudiantes y sus matrículas; países y sus capitales; países y su PIB; palabras y su frecuencia de uso en un documento, etc. También se les llaman “arreglos asociativos”.

Podemos decir que las listas son un tipo de diccionario donde las claves estarían asociadas a los índices, con la condición adicional de que dichas claves son únicas en cada diccionario. No existen elementos repetidos dentro de un diccionario.

Diccionario

keys	values
'b'	'beauty'
'j'	'joy'
'c'	'computing'

*dic* = { 'b': 'beauty', 'j': 'joy', 'c': 'computing' }

clave : valor

Lista

indices	values
0	'beauty'
1	'joy'
2	'computing'

*list* = [ 'beauty', 'joy', 'computing' ]

índice : valor    0       1       2

Por ejemplo, podemos definir un diccionario de 3 pares “clave:valor” (“key:value”) como sigue:

```
dic = {'nombre':'Luis', 'estado':'Tamaulipas', 'edad':23}
dic
{'nombre': 'Luis', 'estado': 'Tamaulipas', 'edad': 23}
```

Podemos acceder cualquiera de los “valores” mediante su “clave” respectiva:

```
dic['estado']
'Tamaulipas'
```

También podemos actualizar cualquier “valor” (value) de una de las “claves” (key):

```
dic['estado'] = 'Nuevo León'
dic
{'nombre': 'Luis', 'estado': 'Nuevo León', 'edad': 23}
```

También podemos incluir nueva información, por ejemplo con un nuevo par “clave:valor”:

```
dic.update({'ZIP':63001})
dic
{'nombre': 'Luis', 'estado': 'Nuevo León', 'edad': 23, 'ZIP': 63001}
```

O actualizar /modificar una de las “claves” (keys):

```
dic['ZP'] = dic.pop('ZIP')
dic
{'nombre': 'Luis', 'estado': 'Nuevo León', 'edad': 23, 'ZP': 63001}
```

Dado un diccionario:

```
dic  
  
{'nombre': 'Luis', 'estado': 'Nuevo León', 'edad': 23, 'ZP': 63001}
```

Podemos acceder sus diferentes entidades como sigue:

```
dic.keys()  
  
dict_keys(['nombre', 'estado', 'edad', 'ZP'])  
  
dic.values()  
  
dict_values(['Luis', 'Nuevo León', 23, 63001])  
  
dic.items()  
  
dict_items([('nombre', 'Luis'), ('estado', 'Nuevo León'), ('edad', 23), ('ZP', 63001)])
```

Se puede también acceder a cada una de las claves o valores de un diccionario de manera individual o iterando con un FOR, ya sea a través de sus “items” o transformándolos en listas. Los siguiente ejemplos muestran cada uno de estos casos:

```
dicc  
  
{'Laura': 8, 'Diana': 7, 'Mario': 8, 'Javier': 9}  
  
lista_keys = list(dicc.keys())  
lista_values = list(dicc.values())
```

Podemos transformar las claves (keys) y valores (values) en listas y manipularlas como tal:

```
lista_keys[3]  
  
'Javier'  
  
lista_values[3]  
  
9  
  
lista_values[0:3]  
  
[8, 7, 8]
```

Podemos también iterar sobre los “items”:

```
for k,v in dicc.items():  
    print(k,":",v)
```

```
Laura : 8  
Diana : 7  
Mario : 8  
Javier : 9
```



Dado el siguiente diccionario:

```
dicc = {'Laura':8, 'Diana':7, 'Mario':8, 'Javier':9}
dicc

{'Laura': 8, 'Diana': 7, 'Mario': 8, 'Javier': 9}
```

Podemos obtener la clave (key) o claves asociadas a un valor (value) particular de varias formas. Dos de estas formas podrían ser las siguientes:

Forma 1:

```
valor = 8

{k for k in dicc if dicc[k]==valor}

{'Laura', 'Mario'}
```

Forma 2:

```
lista_keys = list(dicc.keys())
lista_values = list(dicc.values())

idx = lista_values.index(valor)
lista_keys[idx]

'Laura'
```

En esta segunda forma, si hay más de una clave, habrá que incrementar el índice de búsqueda para encontrar las siguientes claves.



## Vocabularios y Diccionarios:

En particular los diccionarios los utilizaremos para construir el vocabulario de un corpus.

Describamos de manera simple con un ejemplo la manera en que construiremos y usaremos un vocabulario. Por ejemplo, es muy común que la gente opine con mensajes en Twitter sobre alguna película que acaba de salir en cartelera, en particular que indique si le gustó o no le gustó. El tratar de clasificar uno de estos comentarios como positivo o negativo de manera automática usando algún modelo de machine learning, es lo que se llama “análisis de sentimiento”. Esta temática es una de las más estudiadas y de las de mayor interés dentro del área de procesamiento de lenguaje natural.

Resulta que para poder realizar este análisis de análisis de sentimiento, primero debemos construir un “vocabulario” a partir del corpus de entrada. Es decir, a partir del corpus o conjunto de comentarios de Twitter que tengas para analizar, se construye el conjunto de palabras (tokens) que formarán tu vocabulario. Solamente las palabras que formen parte de este vocabulario y diccionario, serán las que existan o utilices para llevar a cabo el análisis y clasificación de los comentarios como positivos o negativos. Por ejemplo, puedes determinar que en tu diccionario solamente tomarás en cuenta palabras (tokens) del vocabulario cuya frecuencia porcentual de aparición en el corpus sea mayor al 2% y eliminar así palabras que sean muy raras o de poco uso.

En el JupyterNotebook de esta semana estudiaremos un ejemplo práctico.

## Conjunto (*set*):

Un **conjunto** considera todos los elementos repetidos como un solo elemento, no toma el orden en el que aparecen y se definen mediante llaves:

```
x = {'m', 'N', 'casa', 'N', 'ropero', 6, 6, 6}
print(x)
```

```
{6, 'N', 'ropero', 'm', 'casa'}
```

x

[🔍] x

set

```
y = {'ropero', 6, 'casa', 'casa', 'm', 'N', 'ropero', 'm', 6}
y
```

```
{6, 'N', 'casa', 'm', 'ropero'}
```

```
x == y
```

```
True
```

Observa que en las listas (definidos por corcheas) sí importa el orden.

```
A = ['x', 'y', 'z']  
B = ['x', 'y', 'z', 'x']  
C = ['z', 'y', 'x']  
print(A, B, C)
```

```
['x', 'y', 'z'] ['x', 'y', 'z', 'x'] ['z', 'y', 'x']
```

```
print(A==B)  
print(A==C)  
print(B==C)
```

```
False  
False  
False
```

Sin embargo, observa que cuando ordenas mediante el método “sorted” a un “conjunto”, este se transforma de manera automática en una “lista”, y por lo tanto ahora el orden sí es importante. Veámoslo con el siguiente ejemplo:

```
q = ['C', 'a', 'l', 'l', 'm', 'e', 'I', 's', 'h', 'm', 'a', 'e', 'l'] # definimos una lista.
print('list:', q)
r = set(q) # transformamos dicha lista en un conjunto.
print('set:', r)
t = sorted(r) # ordenamos los elementos del conjunto y se transforman en una lista.
print('list:', t)
```

```
list : ['C', 'a', 'l', 'l', 'm', 'e', 'I', 's', 'h', 'm', 'a', 'e', 'l']
set : {'a', 'h', 'e', 'C', 'l', 'm', 's', 'I'}
list : ['C', 'I', 'a', 'e', 'h', 'l', 'm', 's']
```

Puedes incluir nuevos elementos a un conjunto con el comando “update”:

```
Q = {'a', 'b'}  
Q.update('c', 'd')  
Q  
  
{'a', 'b', 'c', 'd'}
```

```
Q.update({'w', 'm', 'w'})  
Q  
  
{'a', 'b', 'c', 'd', 'm', 'w'}
```

Observa que puedes actualizar los elementos de un conjunto usando o no paréntesis.

Recuerda que hay muchos más métodos y siempre puedes consultar la documentación respectiva de cada uno de ellos para conocer con mayor detalle la manera en que operan.

En cuanto se vaya requiriendo profundizar en algunos de estos métodos lo haremos:

A.

- ❏ discard
- ❏ intersection
- ❏ intersection\_update
- ❏ isdisjoint
- ❏ issubset
- ❏ issuperset
- ❏ pop
- ❏ remove
- ❏ symmetric\_difference
- ❏ symmetric\_difference\_update
- ❏ union
- ❏ update builtin\_function\_or\_method

## Tuple:

Una “tupla” (“tuple” en inglés) se define mediante paréntesis redondos, a diferencia de una lista que se define mediante corcheas y un conjunto mediante llaves:

```
x = ['i', 'am', 'thor', 'son', 'of', 'odin']  # list
x
['i', 'am', 'thor', 'son', 'of', 'odin']

y = tuple(x)  # tuple
y
('i', 'am', 'thor', 'son', 'of', 'odin')
```

Las listas y las tuplas son muy parecidas, sin embargo se utilizan para diferentes propósitos.

Las principales diferencias entre ambos es que las tuplas requieren menos memoria y por lo tanto son más rápidas de manipular que las listas, además de que pueden ser usadas como “keys” en los diccionarios. Sin embargo, las tuplas son inmutables y tienen menos métodos.



Sobre lo que hemos comentado de que las “tuplas” son inmutables y las “listas” mutables, nos referimos a que una vez definidas, a las listas se les pueden agregar nuevos elementos o quitar algunos, pero no así a las tuplas.

Una vez definida una tupla, esta no puede modificar su tamaño:

```
T = ('a', 'b', 'a')
T
('a', 'b', 'a')
```

Sin embargo, a una lista se le pueden agregar o quitar elementos:

```
L = ['a', 'b', 'a']
L
['a', 'b', 'a']

# podemos agregar elementos:
L.append('c')
L
['a', 'b', 'a', 'c']

# o podemos remover algunos:
L.remove('a')
L
['b', 'a', 'c']
```

En las tuplas podemos seguir accedendo sus elementos como en las listas:

```
t = ('a', 'b', 8, 9, 10)
t
```

```
('a', 'b', 8, 9, 10)
```

```
t[3]
```

```
9
```

```
t[1:4]
```

```
('b', 8, 9)
```

En el caso de que todos los elementos de una tupla o una lista sean números, podemos aplicar algunas funciones matemáticas como las siguientes:

```
X = (7,4,5,6)
print(sum(X))
print(min(X))
print(max(X))
```

```
22
4
7
```

```
Y = [7,4,5,6]
print(sum(Y))
print(min(Y))
print(max(Y))
```

```
22
4
7
```

Sin embargo, recuerda que al ordenarlos con “sorted”, la tupla también se transforma en una lista:

```
Z = sorted(X)
```

```
Z
```

```
[4, 5, 6, 7]
```

X : tupla  
Z : lista

Como mencionamos  
previamente, en las listas  
podemos eliminar  
elementos particulares:

```
L = ['a', 'b', 'c']  
L[1]
```

```
'b'
```

```
del L[1]
```

```
L
```

```
['a', 'c']
```

Sin embargo, en una tupla no puedes  
eliminar un dato en particular, si  
intentas hacerlo obtendrás un error:


```
T = ('a', 'b', 'c')  
T[1]
```

```
'b'
```

```
del T[1]
```

```
-----  
TypeError          Traceback (most recent call last)  
<ipython-input-92-cbf0d8508ae2> in <module>  
----> 1 del T[1]
```

```
TypeError: 'tuple' object doesn't support item deletion
```



Por el momento, si deseas profundizar en algunos de los tipos de datos que hemos estudiado, puedes consultar al menos la siguiente liga:

Diccionario:

<https://docs.python.org/3/tutorial/datastructures.html#dictionaries>

Conjunto (set):

<https://docs.python.org/3/tutorial/datastructures.html#sets>

Tuplas:

<https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences>



## Listas por Comprensión en Matemáticas

### Notación matemática para denotar Conjuntos

Habiendo definido el tipo de dato “set” de Python, retomemos la definición Matemática de lista de comprensión que estudiamos la semana pasada.

En Matemáticas es usual usar la siguiente notación para definir un conjunto, donde se indican las propiedades que deben satisfacer sus elementos:

$$A = \{w \mid w \in U \wedge P(w)\}$$

Lo anterior se lee como sigue:

El conjunto  $A$  está formado por todos los elementos  $w$ , tales que  $w$  pertenece al universo  $U$  y además satisface la propiedad  $P$ .

#### NOTA:

Recuerda que en Matemáticas el símbolo  $\in$  se usa para denotar la pertenencia de un elemento a un conjunto; el símbolo  $\wedge$  denota el operador lógico “y”; el símbolo  $\mid$  se usa simplemente para denotar la expresión “tal que” o “tales que”.

Por ejemplo, definamos matemáticamente el siguiente conjunto mediante la notación de lista de comprensión:

$$A = \{x \mid x \text{ es un entero positivo } \wedge x \text{ divisible por } 3\}$$

Este conjunto se puede representar ahora explícitamente indicando algunos de sus primeros elementos:

$$A = \{3, 6, 9, 12, 15, 18, 21, 24, \dots\}$$

y donde los tres puntos suspensivos al final indican que la cantidad de elementos continúa infinitamente. En Matemáticas es muy común tener conjuntos con una cantidad infinita de elementos, por lo que esta notación es muy práctica para representarlos.

Sin embargo, podemos también definir conjuntos finitos con esta notación añadiendo más condiciones/propiedades. Por ejemplo, podemos ahora definir el siguiente conjunto:

$$B = \{x \mid x \text{ es un entero positivo } \wedge x \text{ divisible por } 3 \wedge x < 20\}$$

donde vemos que B es un conjunto finito, con solo 6 elementos:

$$B = \{3, 6, 9, 12, 15, 18\}$$

Usaremos ahora una notación equivalente para expresar conjuntos y más entidades en Python.

## Listas por Comprensión en Programación (Python)

Notación compacta para Listas, Conjuntos, Diccionarios, etc.

Ya vimos que en Python, con una notación equivalente a la que se usa en Matemáticas, se puede definir un conjunto de elementos de un universo  $U$  y que satisfacen una propiedad  $P$  de manera análoga a las listas de comprensión vista la semana pasada:

$$\{ w \text{ for } w \text{ in } U \text{ if } P(w) \}$$

Por ejemplo, definamos los elementos de nuestro universo  $U$  como un conjunto (set) a partir de los primeros enunciados del poema “Sol de Monterrey” de Alfonso Reyes:

```
U = {'No', 'cabe', 'duda', ':', 'de', 'niño', ',', 'a', 'mí', 'me', 'seguía', 'el', 'sol', '.'}
print(U)
```

```
{',', 'niño', 'No', 'duda', 'cabe', 'seguía', 'a', ':', 'sol', '.', 'de', 'mí', 'me', 'el'}
```

← set

Ahora podemos definir en Python un nuevo conjunto  $A$  que esté formado por elementos del universo  $U$  y que satisfagan la propiedad de que la longitud de dichos elementos sea mayor a tres caracteres:

```
A = {w for w in U if len(w)>3}
print(A)
```

```
{'seguía', 'niño', 'duda', 'cabe'}
```

← set



Sin embargo, ya vimos la semana pasada que la lista de comprensión no se limita solamente al tipo de dato “conjunto” (set).

Veamos nuevamente el ejemplo anterior, pero ahora con una lista:

```
V = ['No', 'cabe', 'duda', ':', 'de', 'niño', ',', 'a', 'mí', 'me', 'seguía', 'el', 'sol', '.']  
print(V)
```

```
['No', 'cabe', 'duda', ':', 'de', 'niño', ',', 'a', 'mí', 'me', 'seguía', 'el', 'sol', '.'] ← list
```

```
A = [w for w in V if len(w)>3]  
print(A)
```

```
['cabe', 'duda', 'niño', 'seguía'] ← list
```

El resultado es parecido al ejemplo del conjunto de la diapositiva anterior, pero ahora el resultado es una lista y aunque los elementos son los mismos, representan entidades diferentes: en particular recuerda que en los conjuntos, el orden de sus elementos no importa y en las listas sí importa.

Recuerda que en Python la estructura general de una lista de comprensión es como sigue:

$$\underbrace{\{ w }_{\text{expresión}} \underbrace{for w in U}_{\text{ciclo FOR}} \underbrace{if P(w) \}_{\text{condición}}$$

Es decir, a través del ciclo FOR estás incluyendo todas las expresiones  $w$  que hacen verdadera la condición  $P$ .

Sin embargo, con dicha notación  $\{w for w in U if P(w)\}$  solo se incluyen elementos que hacen Verdadera la propiedad  $P$ . En caso de tener alguna otra condición para el caso que sea Falsa la condición  $P$ , la sintaxis en Python deberá ser como sigue:

$$\underbrace{\{ w }_{\text{expresión Verdadero}} \underbrace{if P(w)}_{\text{condición}} \underbrace{else Q(w)}_{\text{expresión Falso}} \underbrace{for w in U}_{\text{ciclo FOR}} \}$$

Por ejemplo, consideremos el siguiente diccionario:

```
V = ['No', 'cabe', 'duda', ':', 'de', 'niño', ', ', 'a', 'mí', 'me', 'seguía', 'el', 'sol', '.']  
print(V)  
['No', 'cabe', 'duda', ':', 'de', 'niño', ', ', 'a', 'mí', 'me', 'seguía', 'el', 'sol', '.']
```

Y usemos una lista de comprensión donde cada resultado de la condicional, el Falso y el Verdadero, tienen su expresión particular:

**Expresión  
para el caso  
Verdadero**

condición

**Expresión  
para el  
caso Falso**

ciclo FOR

```
B = [w.capitalize() if len(w)>3 else w.upper() for w in V]  
print(B)  
['NO', 'Cabe', 'Duda', ':', 'DE', 'Niño', ', ', 'A', 'MÍ', 'ME', 'Seguía', 'EL', 'SOL', '.']
```

Toma en cuenta también la manera en que se **inicializa** cada uno de los tipos de datos que hemos visto:

```
L = []    # lista vacía  
L
```

```
[]
```

```
D = {}    # diccionario vacío  
D
```

```
{}
```

```
S = set()  # conjunto vacío  
S
```

```
set()
```

```
T = ()     # tupla vacía  
T
```

```
()
```

## Pandas Series:

Pandas introduce también un tipo de dato llamado “Series”, el cual son arreglos unidimensionales capaces de contener cualquier tipo de dato (enteros, flotantes, strings, etc.).

Una colección de Series, todos con los mismos índices, forman un Data Frame.

Las series son muy parecidas a las listas y podemos transformar los datos de un tipo de dato al otro. Sin embargo, tienen sus diferencias, veamos algunas de ellas:

```
mi_lista = [2,5,7]
mi_serie = pd.Series(mi_lista, index=['A','B','C'])
```

```
print(mi_lista)
print(mi_serie)
```

```
[2, 5, 7]
A      2
B      5
C      7
dtype: int64
```

```
print(mi_lista[0])
print(mi_serie[0])
print(mi_serie['A'])
```

```
2
2
2
```

```
mi_lista * 3
```

```
[2, 5, 7, 2, 5, 7, 2, 5, 7]
```

```
mi_serie * 3
```

```
A      6
B     15
C     21
dtype: int64
```




## Etapa de pre-procesamiento de texto

Recordemos que cualquier modelo de aprendizaje automático (machine learning) requiere recibir datos de buena calidad para generar buenos resultados. A esta etapa usualmente se le conoce como de preparación o pre-procesamiento de los datos.

Esta etapa generalmente está formada por una gran variedad de pasos que no necesariamente se aplican a todos los problemas por igual o se aplican en el mismo orden. La experiencia que vayas ganando como científico/analista de datos te ayudará a ir construyendo los pasos que consideres te llevan a generar los datos más limpios y de mayor calidad, listos para alimentar los modelos de aprendizaje automático (machine learning).

Veamos algunos de los pasos que usualmente forman parte de esta etapa de preparación de los datos de texto.

Uno de los problemas principales en el análisis de texto, es que la cantidad de palabras (o tokens, de manera más general) que pueden formar parte de un diccionario pueden llegar a ser de decenas o de cientos de miles, si no es que de millones. Esta gran cantidad de tokens no solo será un problema por los mayores recursos de cómputo requeridos, sino que los modelos también requerirán una mayor cantidad de datos para llegar a aprender todos sus pesos.





## Stopwords



Las “stopwords” son aquellas palabras que no brindan mayor información en el proceso de entender el significado de un enunciado.

En principio son todas aquellas palabras que aparecen en cualquier documento de texto, sin importar la temática, como los artículos, conjunciones, preposiciones y adverbios.


Así, en ciertos problemas como los llamados de análisis de sentimiento (sentiment analysis), se pueden omitir las stopwords y concentrarse en las palabras que dan mayor valor para el entendimiento “positivo” o “negativo” de un enunciado.

Eliminar las stopwords ayuda también a generar diccionarios más pequeños que requieren menos recursos de cómputo.

En español seguiremos haciendo referencia a este grupo de palabras como “stopwords”, aunque a veces se les llama en español como “palabras vacías”.







Obviamente cada idioma tiene su propio conjunto de stopwords y cada librería también define los suyos, por lo que no dejes siempre de ver las palabras que forman parte de este conjunto. Por otro lado, todas las librerías nos permiten agregar o eliminar palabras de dicho conjunto de stopwords, en caso que así lo consideremos adecuado.

Sin embargo, veamos algunas de las palabras que usualmente aparecen en las stopwords:

#### **Artículos**

el, la, los, las, un,  
una, unos, unas

#### **Conjunciones**


y, ni, o, que, porque, pero,  
sino, mas, aunque, salvo,  
excepto, pues, etc.

#### **Proposiciones**

a, ante, con, contra,  
de, desde, en, entre,  
hacia, hasta, para,  
por, según, sin, sobre,  
versus, etc.

#### **Adverbios**

cuando, como,  
donde, aquí, allá,  
no, nunca, jamás,  
tampoco, menos,  
mucho, poco, etc.



Observa que dentro de los adverbios aparecen muchas de las palabras que usamos para frases con “sentido negativo”, esto es algo que retomaremos más adelante al estudiar problemas de análisis de sentimiento.

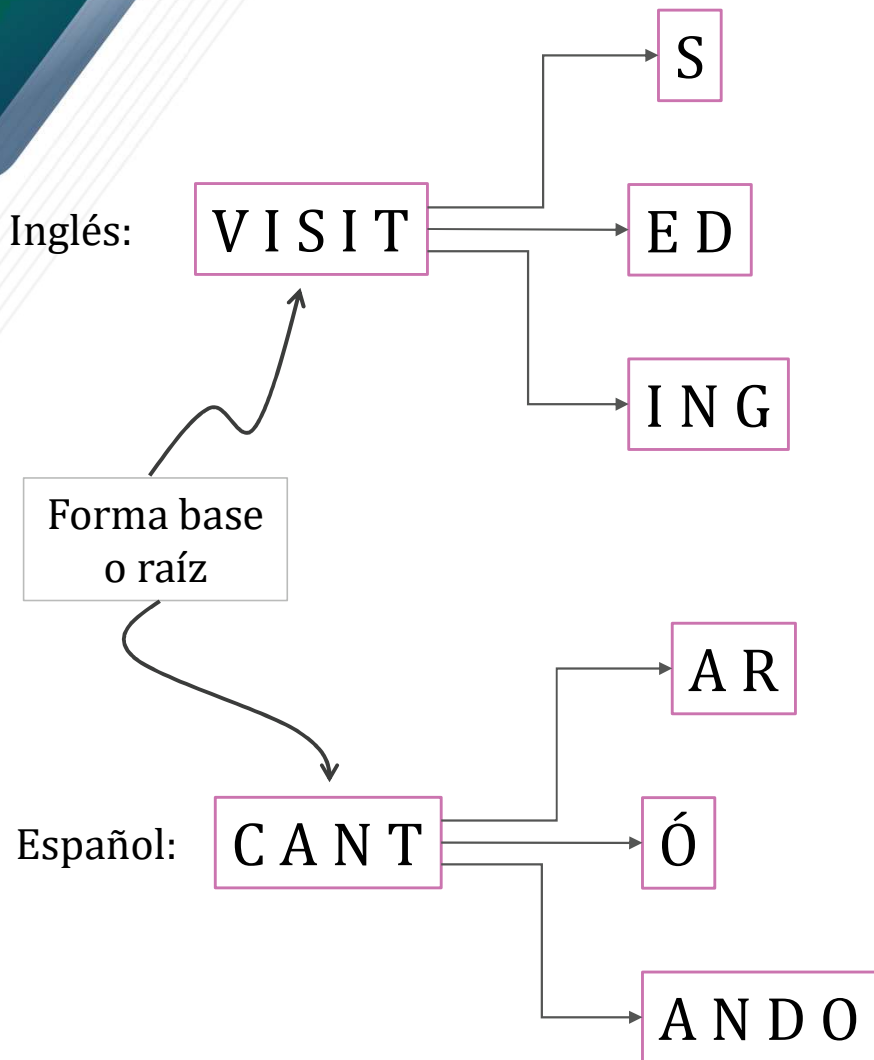


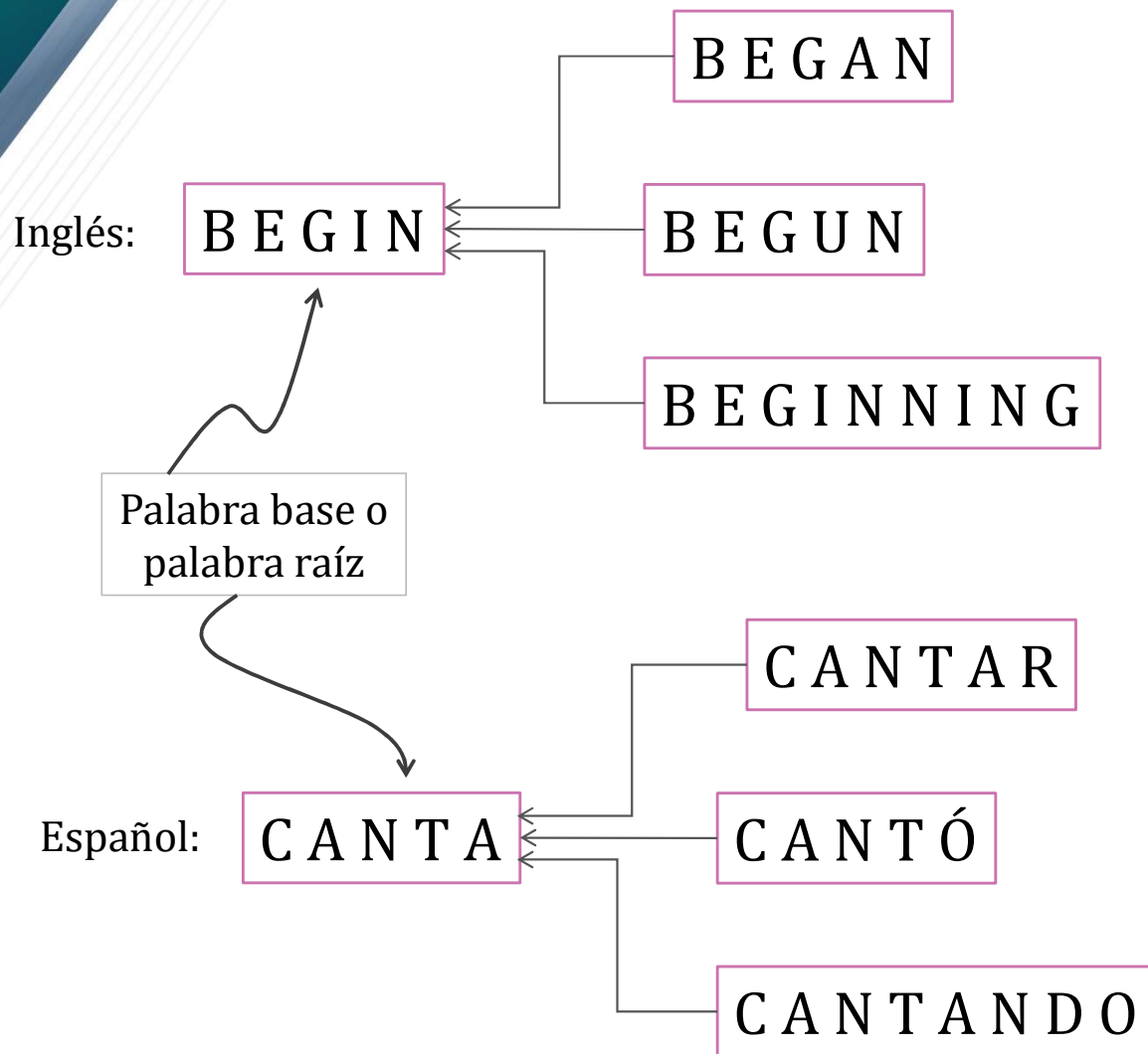
## Stemming

En el análisis de texto en ocasiones se requiere reconocer solamente la **forma base o raíz** de una palabra. Por ejemplo, en el caso de un verbo, no importaría su conjugación o tiempo; para un adjetivo o sujeto no se requeriría saber si está en singular o plural, etc.

Lo anterior ayudará a simplificar el vocabulario a utilizar y por lo tanto también simplificará el análisis y significado de los enunciados. Esto obviamente puede generar otros problemas, pero en muchos análisis es más que suficiente.

Hay que tomar en cuenta que la forma base obtenida con *stemming* no necesariamente es correcta desde el punto de vista gramatical, es decir, no tiene que ser necesariamente una palabra válida del diccionario.





## Lemmatization

Es un proceso análogo al Stemming, donde el string de entrada se sustituye por una **palabra base** o **palabra raíz** llamada **lemma**. La diferencia es que en este proceso de lemmatization, dicha forma base sí debe ser una palabra válida dentro de dicho idioma.


Esto implicará mucho mayor tiempo de procesamiento durante el análisis, ya que se tiene que estar verificando la existencia de dicha palabra en el idioma particular, además del significado sintáctico y semántico de dicha palabra.




Existe una gran variedad de librerías para el proceso de tokenización y procesamiento de texto.

En los archivos de Jupyter Notebook de cada semana estaremos abordando varias de estas librerías y todas tienen sus ventajas y desventajas. Sin embargo, recuerda que el objetivo principal de esta etapa es el poder preparar los documentos de texto de la mejor manera para su ingreso a alguno de los modelos de aprendizaje automático (machine learning) o de aprendizaje profundo (deep learning).

La manera de proceder en la limpieza y preparación de los documentos de texto puede ser muy diversa, pero podemos esquematizar los pasos principales como sigue:



Se irán incorporando otras etapas más adelante y el orden de estos pasos no necesariamente debe ser el que se ilustra, pero te dará una idea de cómo irte adentrando en la etapa de pre-procesamiento de textos.





D.R.© Tecnológico de Monterrey, México, 2022.  
Prohibida la reproducción total o parcial  
de esta obra sin expresa autorización del  
Tecnológico de Monterrey.