Maestría en Inteligencia Artificial Aplicada (MNA)

Tipos de Datos y Expresiones Regulares (regex)

Procesamiento de Lenguaje Natural (NLP)

Luis Eduardo Falcón Morales

Reconociendo las entidades a procesar: Tokenización

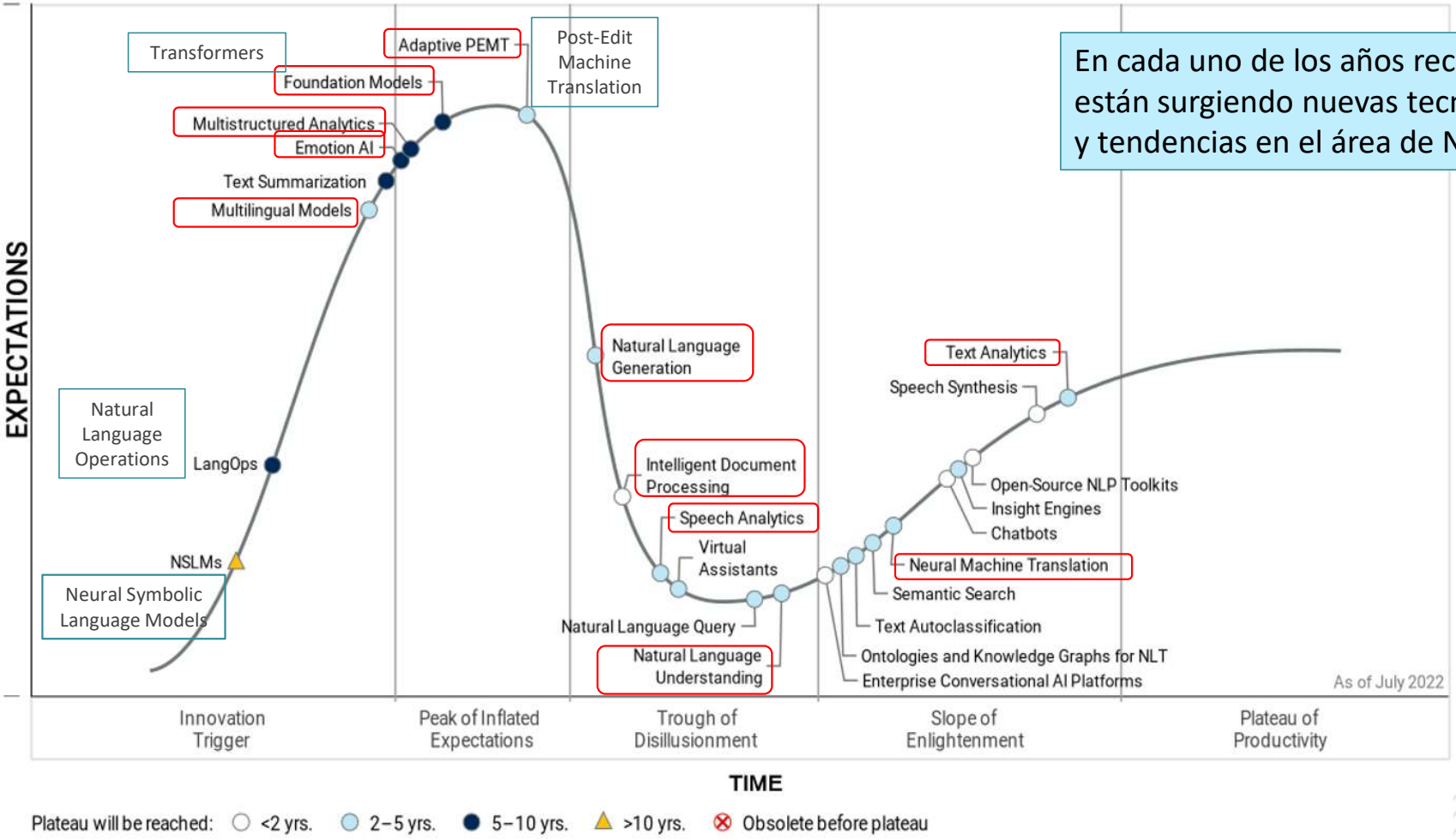
En esta semana estudiarás las unidades mínimas de información que deberás generar para iniciar la extracción de información de un conjunto de documentos.

Estas unidades mínimas de información son conocidas usualmente como “tokens” y al proceso de llevar a cabo dicho proceso, se le conoce como “tokenización”.

Para su manipulación y preparación se requerirá del uso de las llamadas expresiones regulares.

https://www.alkymi.io/library/gartner-hype-cycle-report?utm_campaign=2022%20Social%20General&utm_content=216445320&utm_medium=social&utm_source=twitter&hss_channel=tw-889488813797650432

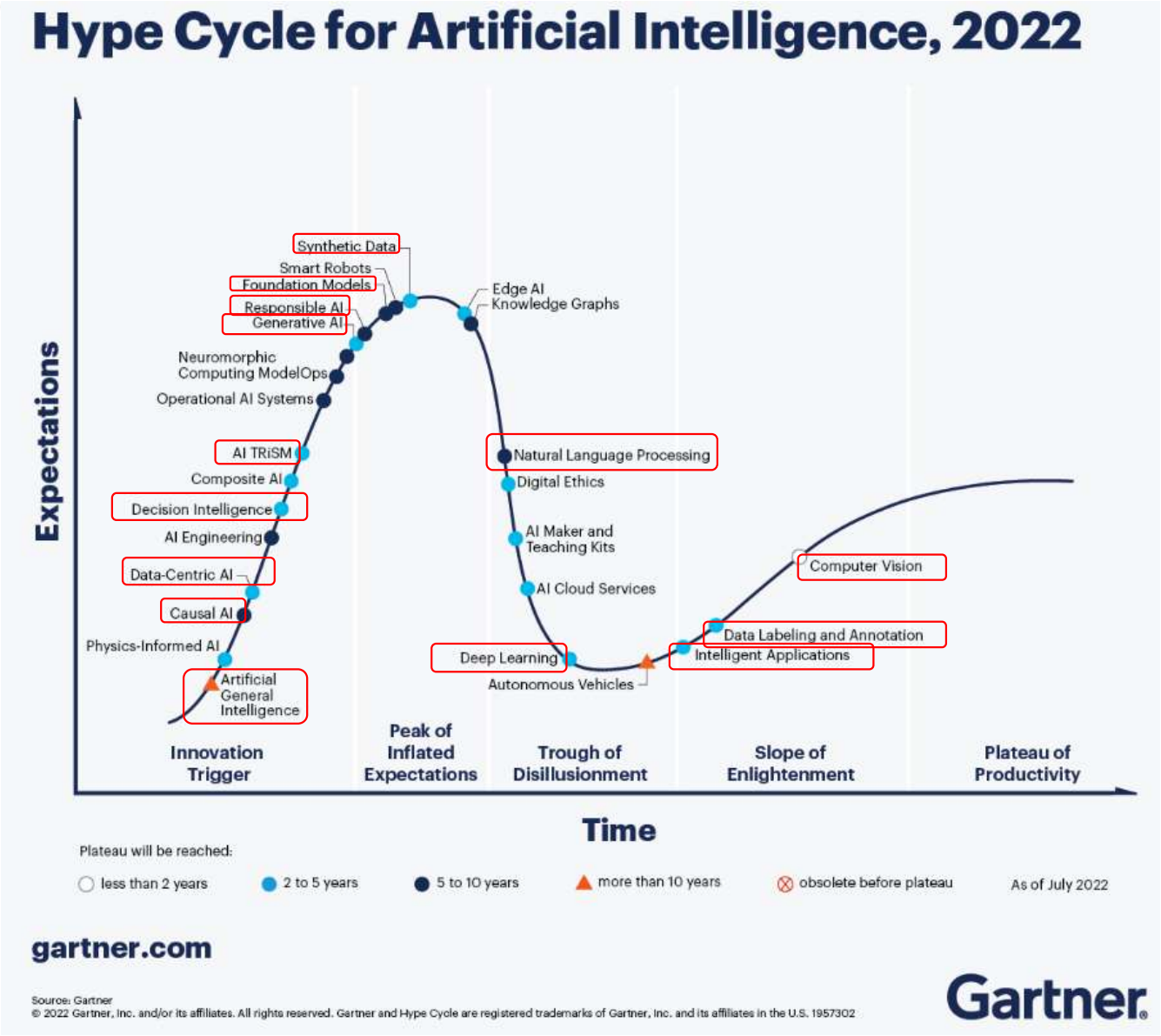
Hype Cycle for Natural Language Technologies, 2022



Plateau will be reached: ○ <2 yrs. ● 2–5 yrs. ● 5–10 yrs. ▲ >10 yrs. ⊗ Obsolete before plateau

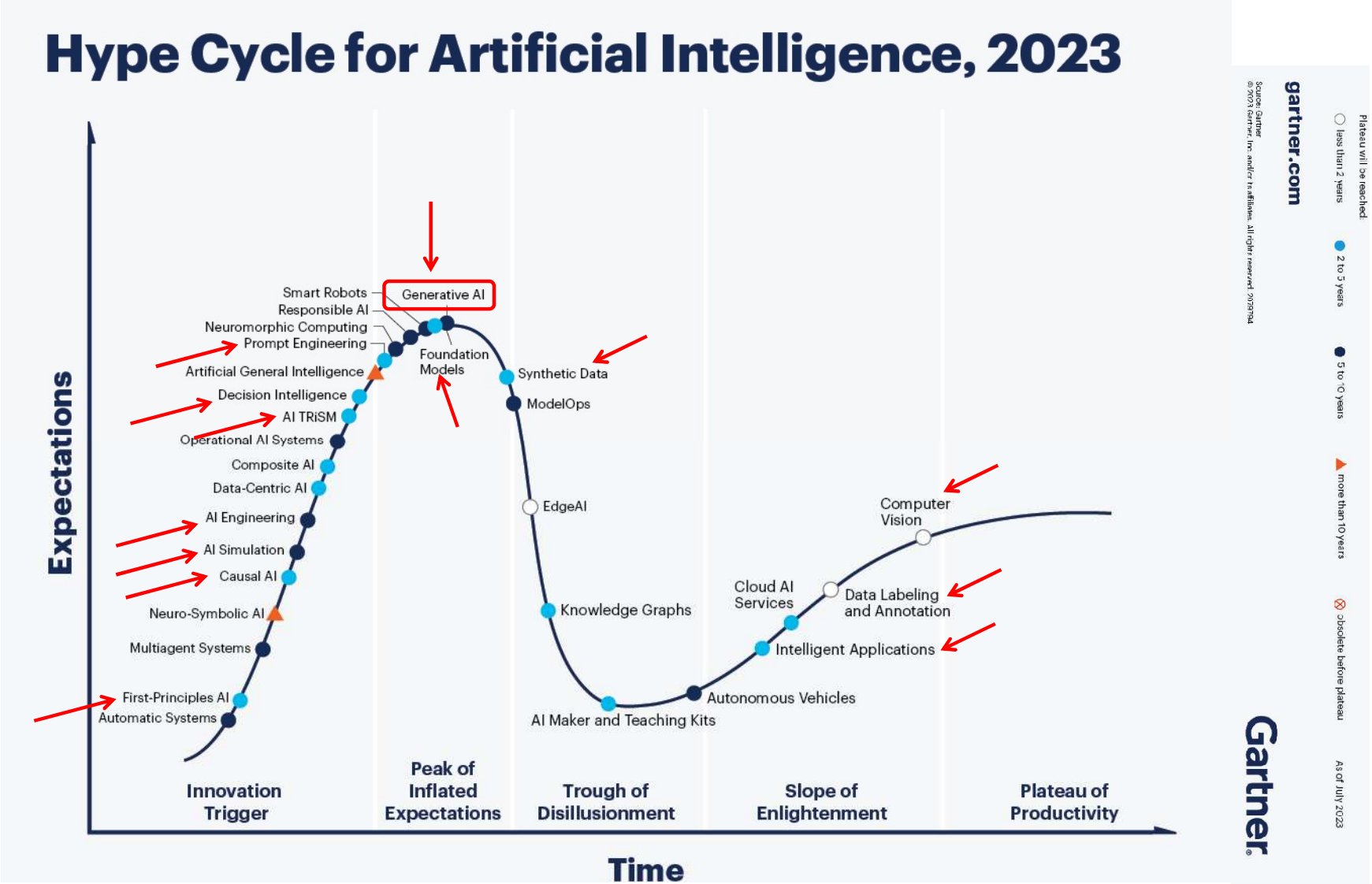
En cada uno de los años recientes están surgiendo nuevas tecnologías y tendencias en el área de NLP.

https://www.gartner.com/en/articles/what-s-new-in-artificial-intelligence-from-the-2022-gartner-hype-cycle



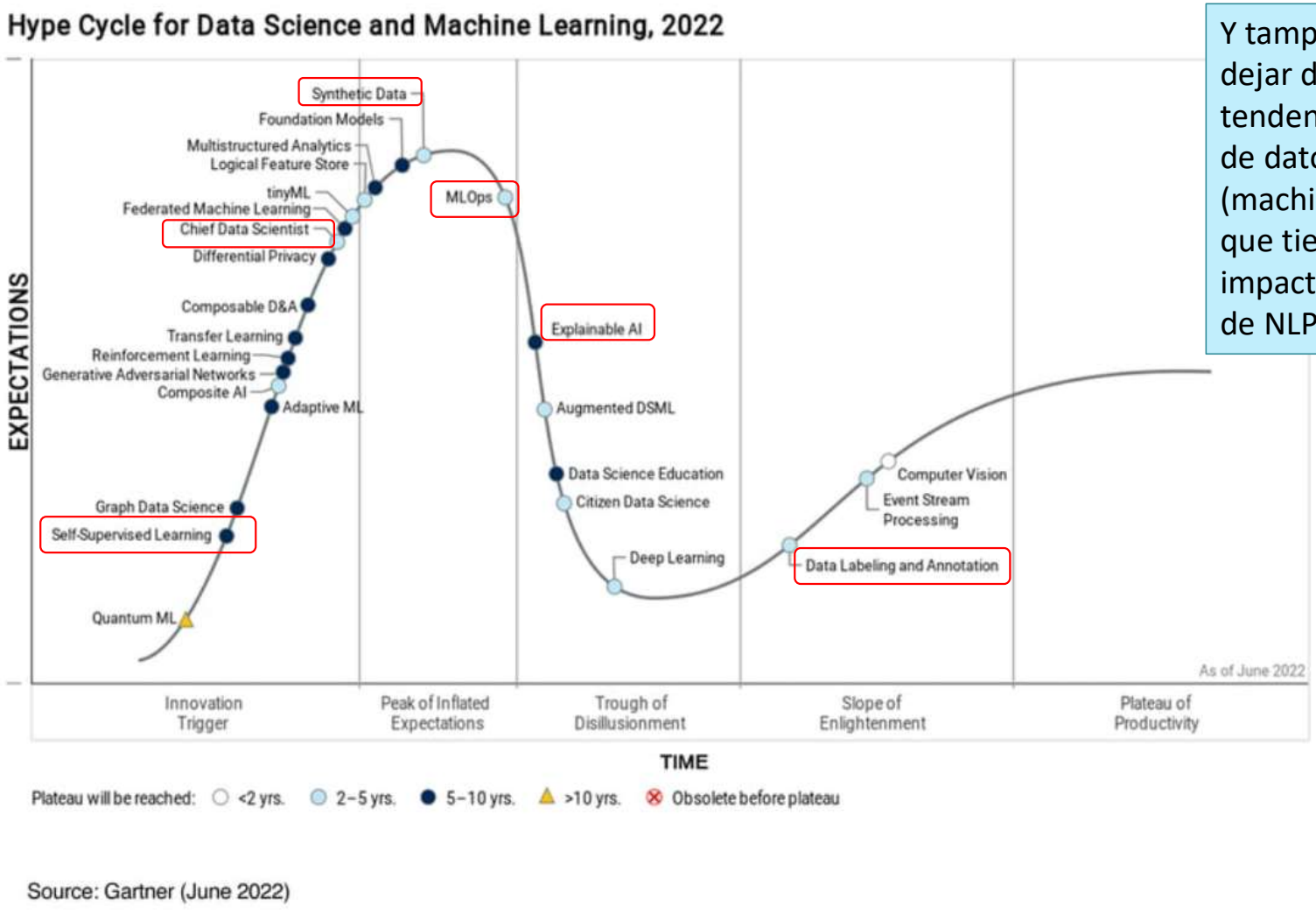
Las tendencias en el área de NLP están obviamente muy ligadas a las tendencias de IA en general.

Gartner indica que actualmente casi el 100% de las empresas de cualquier tamaño desean utilizar la IA, pero solo cerca del 10% lo están llevando a producción.



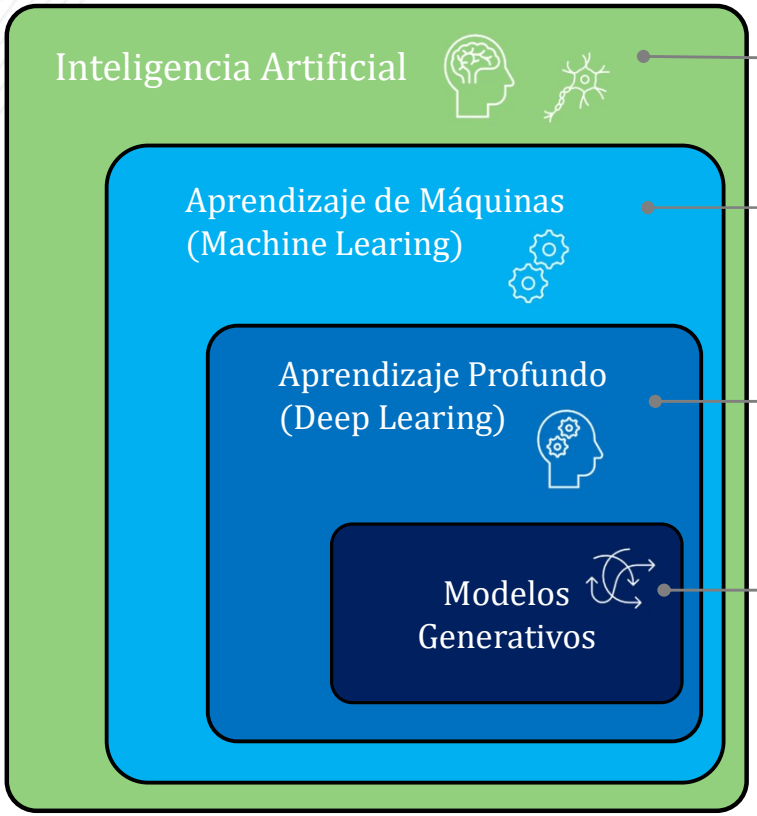
https://www.gartner.com/en/articles/what-s-new-in-artificial-intelligence-from-the-2023-gartner-hype-cycle

https://pages.dataiku.com/gartner-hype-cycle-dsml



Source: Gartner (June 2022)

Y tampoco podemos dejar de lado las tendencias en ciencia de datos y ML (machine learning) que tienen gran impacto en los tópicos de NLP.



$1950's$

Inteligencia Artificial

Cualquier método que permita a una computadora simular el comportamiento humano.

$1980's$

Aprendizaje de Máquinas (Machine Learning-ML)

Área de la IA que aplica técnicas diversas para que las computadoras mejoren su desempeño con base a la experiencia.

$2010's$

Aprendizaje Profundo (Deep Learning-DL)

Área de ML que implementa modelos de redes neuronales profundas.

$2020's$

Modelos Generativos (Generative models)

Área de DL que genera datos de manera probabilística con base al conocimiento de la distribución de los datos.

Inteligencia Artificial (IA)

- Aprendizaje Automático
  - Aprendizaje supervisado
  - Aprendizaje no supervisado
  - Aprendizaje por refuerzo
  - Aprendizaje profundo
- Procesamiento de Lenguaje Natural
  - Traducción automática
  - Extracción de contenido
  - Preguntas & Respuestas
  - Generadores de texto
- Visión Computacional
  - Reconocimiento de imágenes
  - Video análisis
  - Segmentación
  - Reconocimiento biométrico
- Audio
  - Audio a texto
  - Texto a audio
- Robótica
- Optimización
- Sistemas Expertos
- ⋮

Para llegar a cumplir estos objetivos generales de NLP debemos empezar de la manera usual a cualquier problema de ML, con el preprocesamiento de los datos, en este caso, con el preprocesamiento de los documentos de texto.

Los algoritmos de aprendizaje automático requieren entradas numéricas, ya que procesan la información mediante operaciones matemáticas.

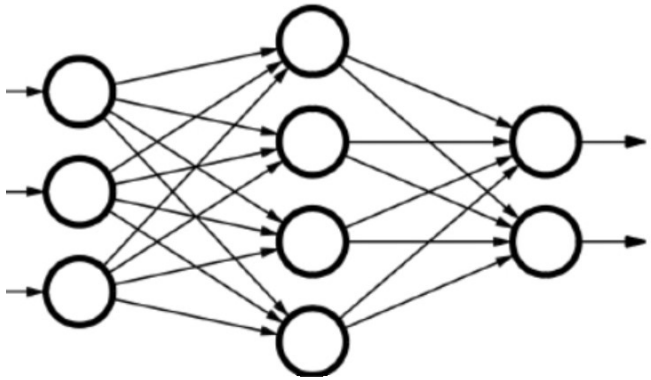
Los datos estructurados históricamente han sido los que impulsaron esta área de estudio, debido a su relativa facilidad de transformarlos a un formato numérico que sea entendible por los algoritmos de aprendizaje automático.

Datos estructurados:

Es relativamente sencillo obtener una representación numérica de estos.

Algoritmo de aprendizaje automático (machine learning)

Entrada: información numérica



Salida: Clasificación / Predicción

Sin embargo, ¿cómo proceder con datos no estructurados como las imágenes o los documentos de texto? Es decir, ¿cómo representar estos datos no estructurados en un formato numérico que pueda ser procesado con operaciones matemáticas?

Texto : Datos no estructurados

Dos de los principales datos no estructurados que en las últimas décadas han ayudado a desarrollar el campo de la inteligencia artificial en general y del aprendizaje profundo (deep learning) en particular, son las imágenes y el texto.

Las imágenes y el video han ayudado a expandir el área de visión computacional. El formato estándar de una imagen permite asociarlo rápidamente al concepto de “matriz” que se usa en Matemáticas, lo cual permitió que el área de visión computacional tuviera un crecimiento exponencial en las últimas décadas.

Por su parte, los documentos de texto pueden transformarse rápidamente a un formato numérico mediante la transformación de cada caracter a su código ASCII o alguna de sus extensiones. Sin embargo, debido a la complejidad que implica el entendimiento de un documento de texto, esta transformación mediante el código ASCII es un formato muy ineficiente, que no permite extraer toda la información subyacente contenida en los documentos de texto. La búsqueda de un método más eficiente para transformar un texto a información numérica relevante, ha requerido de muchos más años de estudio. Sin embargo, una vez propuesta una representación numérica vectorial del texto más eficiente en el 2013 por parte de un grupo de investigadores de Google, el desarrollo de NLP se ha potencializado con una gran cantidad de aplicaciones. En particular, el gran impacto mediático que están teniendo aplicaciones como chatGPT-3 y DALL-E en el último año, es un pequeño ejemplo que muestra que las aplicaciones basadas a partir de datos no estructurados de texto, han llegado para cambiar la forma en que hacemos muchas de las tareas actuales.

¿Qué es una Imagen Digital?

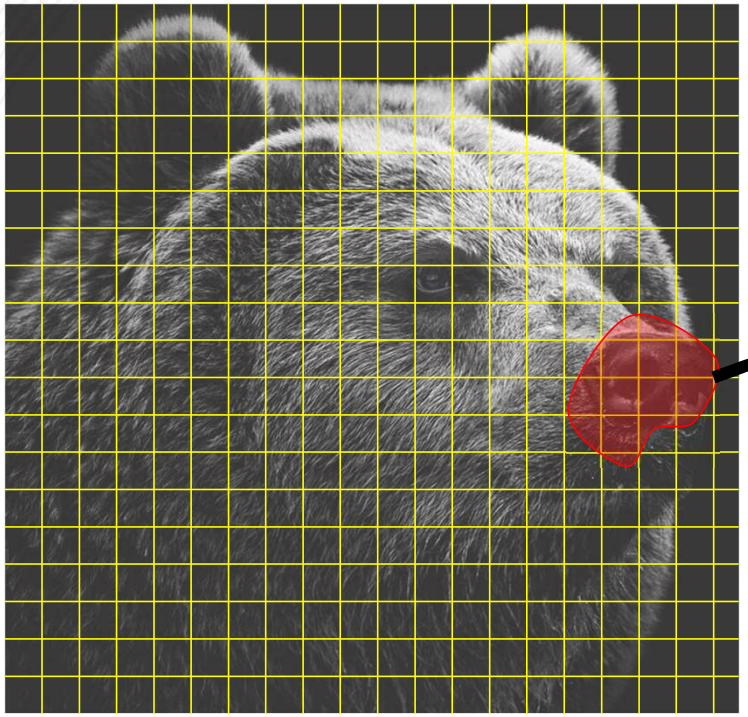


Imagen de grises de 8 bits :
$2^8 = 256$ tonos de gris

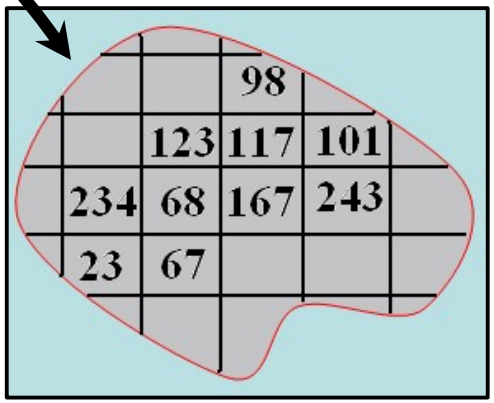
Tamaño de imagen:
$512 \times 512 = 262{,}144$ pixeles

Matriz o Arreglo Bidimensional:

La imagen digital como un arreglo matricial:

Matriz

$$\text{Imagen} = A(k, j) = \left(a_{k,j}\right)_{M \times N}$$



Valor de los pixeles
(picture – elements)

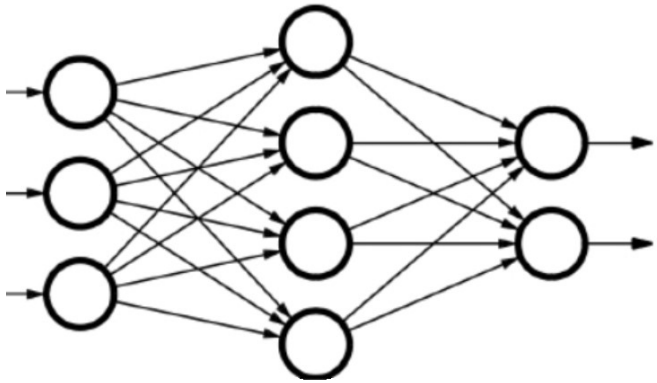
Una imagen en tono de grises es una arreglo rectangular de enteros (por ejemplo de 0 a 255). Este arreglo rectangular es muy parecido al concepto de "matriz" que se estudia en Álgebra Lineal. Y por ello, aplicar transformaciones a una imagen mediante "máscaras" para extraer su información relevante, fue una tarea bastante directa.
Esta relación más cercana de las imágenes con las matrices llevaron a desarrollar más rápidamente esta área que la de textos.

Texto:
Datos no estructurados

¿¿ cómo transformarlos ??

Ahora, la pregunta que surge es ¿cómo encontrar una representación numérica de los documentos de texto para que los algoritmos de aprendizaje automático puedan extraer toda la información que contienen?

Entrada:
información numérica



Salida:
Clasificación / Predicción / Generación de texto ...

Actualmente el área de estudio de NLP es de gran interés y desarrollo de nuevas soluciones ya que sus áreas de aplicación son muy diversas, entre las que podemos mencionar:

- Resumen de textos
- ChatBots
- Análisis de sentimientos
- Traducción automática entre diferentes idiomas
- Clasificación automática de textos
- Generación de texto en alguna temática particular
- Preguntas y respuestas
- Etc..

Sin embargo, para enfrentar los problemas anteriores, primero debemos resolver algunos retos que nos permitan procesar y entender un enunciado:

- Conocimiento de todas las palabras de un diccionario.
- Poder incorporar palabras nuevas o tokens (por ejemplo, emojis) al diccionario.
- Determinar qué hacer con las palabras que van quedando en desuso.
- El significado de una palabra puede ser múltiple (depender del contexto).
- El significado de una palabra puede cambiar con el tiempo.
- Entender los modismos o regionalismos.
- Entender el uso de símbolos, abreviaciones, ideogramas, emojis, etc.
- Entender las palabras que no respetan las reglas de ortografía.
- Entender enunciados de un idioma, que intercalan palabras en otro idioma.
- etc …

En el proceso de transformar y obtener información numérica a partir de documentos de texto, se requiere primero procesar cada enunciado como tal, así como cada una de sus unidades mínimas que la componen. Es decir, debemos entender primero cómo manipular cada carácter, letra o palabra de manera individual, antes de querer analizar toda una oración.

Así, debemos establecer primero cuáles serán las unidades mínimas de información a extraer de un documento de texto, a partir de las cuales se iniciará el proceso de análisis y entendimiento del mismo.

Este análisis inicial suele llevarse a cabo mediante el proceso llamado "tokenización", que estudiaremos a continuación.

Tokenización

Llamaremos “tokenización” al proceso de separar un enunciado en sus entidades mínimas o individuales de información y que llamaremos tokens.

En un texto, dichos tokens pueden definirse como una de las letras o caracteres individuales, o bien como cada una de las palabras de un enunciado, o conjunto de palabras (n-gramas). No existe una manera única de hacerlo y la decisión dependerá del analista y del problema que se está intentando resolver.

Por simplicidad, iniciaremos considerando a las palabras mismas de un enunciado como las unidades mínimas de información (tokens), así como algunos signos de puntuación.

Se deberá especificar cuáles signos de puntuación se considerarán parte de los tokens.

Igualmente se pueden incluir algunos caracteres especiales como parte de los tokens.

Se debe especificar si habrá diferencia entre letras mayúsculas y minúsculas, dígitos, fechas, etc.

El proceso de tokenización nos ayudará a procesar, analizar y entender mejor los enunciados.

Por ejemplo, podemos generar tres tipos de tokenización en la siguiente frase de Federico Nietzsche:

Sin música, la vida sería un error.

1) Tokenizando cada palabra como tal y con los signos de puntuación:

“Sin”, “música”, “,”, “la”, “vida”, “sería”, “un”, “error”, “.”

Se tienen 9 tokens.

2) Tokenizando cada palabra como tal, pero sin signos de puntuación:

“Sin”, “música”, “la”, “vida”, “sería”, “un”, “error”

3) Tokenizando en minúsculas, sin acentos y sin signos de puntuación:

“sin”, “musica”, “la”, “vida”, “seria”, “un”, “error”

Se tienen 7 tokens.

Así, en Español, al igual que en otros idiomas, deberá especificarse si se consideran los acentos u otros caracteres especiales como parte de los tokens.

Así, empezaremos a trabajar con tokens, pero ¿cómo manipulamos cada palabra o token en una computadora?

La manera estándar de representar una palabra (token) en una computadora es mediante el tipo de dato llamado “string”, o cadena de caracteres, en español.

Sin embargo, veremos que dichos strings pueden agruparse de formas diferentes, dando a su vez diferentes tipos de datos.

Será importante conocer cada uno de estos tipos de datos en los cuales se puede capturar información de texto, porque ello implicará diferentes formas de procesarlo o transformarlos.

Veamos a continuación los principales tipos de datos en los cuales podremos empezar a capturar información de un texto.

Tipos de datos

Estudiaremos a continuación los siguientes tipos de datos que nos permitirán manejar diferentes estructuras de datos que aparecerán comúnmente en el manejo de documentos de texto. Podemos decir que una estructura de datos es una colección de datos de diferentes tipos. Entre los principales tipos de datos podemos mencionar los siguientes:

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

Datos numéricos (numeric) & Booleanos (bool)

Suponemos que ya estás bastante familiarizado con este tipo de datos.

Recordemos simplemente que entre los más conocidos que estaremos utilizando en el lenguaje de programación Python y a lo largo del curso son, de manera muy general: booleanos, enteros y flotantes.

Booleano (bool):
True / False

```
b = True
True                bool
```

Entero (int):

```
n = 23
n
n                   int
```

flotante (float):

```
x = 2.71
x
x                   float
```

Por el momento con esta información es suficiente y no haremos diferencia entre los diferentes tipos de representación de los enteros y flotantes. En el momento que se requiera hacer mayor énfasis, lo haremos.

Cadena de caracteres (*string*):

Una “cadena de caracteres” o “string” es una secuencia de caracteres entre comillas, sencillas o dobles.

Así, un enunciado se podrá representar mediante un string. Por ejemplo con las frases que inicia la novela Pedro Páramo de Juan Rulfo:

```
doc = "Vine a Comala porque me dijeron que acá vivía mi padre, un tal Pedro Páramo. Mi madre me lo dijo."

doc
```

```
doc                                str
```

Podemos accesar cada uno de los caracteres del string de manera individual:

```
doc[3]
```

```
'e'
```

```
doc[-1]
```

```
'.'
```

O bien podemos obtener una secuencia de ellos:

```
doc[:9]
```

```
'Vine a Co'
```

```
doc[-8:-1]
```

```
'lo dijo'
```

Observa que cada caracter de un string se puede etiquetar con enteros positivos o negativos, siendo el primero de ello, el de etiqueta cero:

0	1	2	3	4	5	6	7	8	9
H	o	l	a		m	u	n	d	o
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
u = "Hola mundo"
```

```
print(u[0])
print(u[1])
print(u[2])
print(u[4])
print(u[6])
print(u[9])
```

```
H
o
l

u
o
```

```
print(u[0:4])
print(u[5:10])
print(u[2:7])
print(u[:3])
print(u[7:])
```

```
Hola
mundo
la mu
Hol
ndo
```

```
print(u[-1])
print(u[-5])
print(u[-8:-3])
```

```
o
m
la mu
```

NOTA: Observa que la referencia [m : n] indica la secuencia del caracter “m” al caracter “n–1 ”.

Algunas operaciones con strings:

```
x = "Hola"
y = "mundo"
```

Podemos sumar strings:

```
w = x + y
print(w)
```

```
Holamundo
```

Pero tome en cuenta que en ocasiones deberás incluir el espacio en blanco:

```
z = x + ' ' + y
print(z)
```

```
Hola mundo
```

Podemos obtener también la “longitud” (length) o cantidad de caracteres de un string:

```
print(len(x))
print(len(y))
print(len(w))
print(len(z))
```

```
4
5
9
10
```

A lo largo del curso iremos viendo más operaciones con strings, en combinación con otro tipo de datos.

```
A = "manzana, pera, naranja, pera, uva."
```

```
A.index('pera')
```

```
9
```

```
A.index('pera', 10)
```

```
24
```

Puedes utilizar el método “index” para encontrar el índice de donde inicia una cadena de caracteres particular.

Si hay varias cadenas repetidas, te entrega el índice de donde inicia la primera coincidencia.

Puedes indicarle ahora que busque a partir del siguiente caracter encontrado de la primera coincidencia para que encuentre dónde inicia la siguiente coincidencia.

Si ya no existen coincidencias genera un error.

método **split()** de un String

Como hemos comentado previamente, las palabras pueden ser las unidades mínimas de información a partir de las cuales nos proponemos entender el significado de una oración.

El método split() nos permite separar las entidades de un string por sus espacios en blanco: en este caso separamos por las palabras de un enunciado.

```
q = "Benito Juárez nació el 21 de marzo de 1806."
```

```
q.split()
```

```
['Benito', 'Juárez', 'nació', 'el', '21', 'de', 'marzo', 'de', '1806.']
```

Métodos asociados a un String

```
q = "Benito Juárez nació el 21 de marzo de 1806."
```

```
q.upper()
```

```
'BENITO JUÁREZ NACIÓ EL 21 DE MARZO DE 1806.'
```

Además del split(), existen diversos métodos asociados a un String.

Los iremos estudiando a lo largo del curso, pero mientras tanto no dudes en ir experimentando con ellos.

Aquí mostramos un par de ejemplos.

```
for w in q.split():
  print(w, ": ", w.isdecimal())
```

```
Benito :  False
Juárez :  False
nació :  False
el :  False
21 :  True
de :  False
marzo :  False
de :  False
1806. :  False
```

El "." hace que no lo considere "decimal"

```
q.
```

- isdecimal — q.isdecimal()
- split
- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format

Lista (*list*):

La manera usual de expresar un texto o documento es mediante el tipo de dato llamado “lista”.

Una lista se define mediante corcheas.

En el caso de documentos de texto, una lista podrá estar expresado por una secuencia de uno o más strings.

```
w = ['Moby Dick: Call me Ishmael.']
w
w                                                    list
```

```
u = ['Moby', 'Dick', ':', 'Call', 'me', 'Ishmael', '.']
u
u                                                    list
```

```
q = ['M','o','b','y','D','i','c','k',':','C','a','l','l','m','e','I','s','h','m','a','e','l']
q
q                                                    list
```

Cada una de estas 3 variables es una “lista”, sin embargo cada una está formada por diferente cantidad de elementos string.

```python
w = ['Moby Dick: Call me Ishmael.']
```

```python
print(len(w))
print(w[0])
```

```
1
Moby Dick: Call me Ishmael.
```

```python
print(w[0][0:12])
```

```
Moby Dick: C
```

La variable “w” es una lista formada por un solo string.

```python
u = ['Moby','Dick',':','Call','me','Ishmael','.']
print(len(u))
print(u[3])
```

```
7
Call
```

```python
print(u[5][0:6])
```

```
Ishmae
```

La variable “u” es una lista formada por 7 elementos, donde cada uno de estos es un string.

```
x = ['uno','dos']
y = ['dos','uno']

x==y
```

```
False
```

```
u = [1,2]
w = [2,1]

u==w
```

```
False
```

Toma en cuenta que en una lista el orden de los elementos sí es importante.

Operaciones básicas con listas en Python

Tamaño de la lista:

```
u = ['Moby', 'Dick', ':', 'Call', 'me', 'Ishmael', '.']
len(u)
```

```
7
```

Podemos concatenar listas:

```
['Moby','Dick',':'] + ['Call','me','Ishmael','.']
```

```
['Moby', 'Dick', ':', 'Call', 'me', 'Ishmael', '.']
```

Podemos agregar palabras nuevas (appending):

```
u = ['Call', 'me']
u.append('Ishmael')
u
```

```
['Call', 'me', 'Ishmael']
```

Recuperando/manejando elementos de una lista mediante índices:

```
q = ['C','a','l','l','m','e','I','s','h','m','a','e','l']
print( q[4] )
print( q[6:] )
print( q.index('l') )
```

```
m
['I', 's', 'h', 'm', 'a', 'e', 'l']
2
```

Cuando hay elementos repetidos, “index” regresa el índice del primero que encuentra.

Ordenar los elementos de una lista:

```
u = ['Moby','Dick',':','Call','me','Ishmael','.']
sorted(u)
```

```
['.', ':', 'Call', 'Dick', 'Ishmael', 'Moby', 'me']
```

Los ordena iniciando con caracteres especiales, seguido de palabras en orden alfabético, empezando con las que inician con mayúscula.

Las listas tienen también varios métodos que iremos viendo a lo largo del curso. Puedes consultar la documentación correspondiente e ir investigando para qué y cómo funciona cada uno de estos métodos.

```
P

['Benito', 'Juarez', 'nacio', 'el', '21', 'de', 'marzo', 'de', '1806.']

P.
```

- append — P.append(object, /)
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

Cadena de caracteres (*string*) y listas:

Al trabajar con documentos de texto, veremos que los strings y las listas están muy relacionadas.

Podemos conjuntar varias palabras (cada una un string) de una lista *u*, para generar un solo string *w*, mediante el separador que se indique. Por ejemplo, para conjuntarlas separando cada palabra por un espacio en blanco, sería como sigue:

string:

```
x = 'Moby'
x[:3]
```

```
'Mob'
```

list →
string →

```
u = ['Moby','Dick',':','Call','me','Ishmael','.']
w = ' '.join(u)
w
```

Entrada: Lista con 7 elementos.

string →

```
'Moby Dick : Call me Ishmael .'
```

Salida: un solo string.

Y podemos de nuevo separar las palabras del *string w* anterior, para obtener de nuevo la lista de la que partimos:

list →

```
w.split()
```

```
['Moby', 'Dick', ':', 'Call', 'me', 'Ishmael', '.']
```

Lista con 7 elementos.

Listas por Comprensión en Programación (Python)

Notación compacta para Listas, Conjuntos, Diccionarios, etc.

En Python, con una notación equivalente a la que se usa en Matemáticas, se pueden seleccionar elementos de un universo U que satisfacen una propiedad P. En particular, para generar una lista tendríamos lo siguiente:

$$[\, w \;\; for \;\; w \;\; in \;\; U \;\; if \;\; P(w) \,]$$

Observa la estructura general de una lista de comprensión:

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

Es decir, a través del ciclo FOR, estarás incluyendo todas las expresiones que satisfagan la propiedad o condición dada P. En la “expresión” también puede aplicarse alguna otra propiedad a “w”.

Veamos el siguiente ejemplo con las primeras palabras del poema "Sol de Monterrey" de Alfonso Reyes:

```
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

expresión ciclo FOR condición

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

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

← list

En este caso, se seleccionaron las palabras con longitud mayor a 3 caracteres.

Por último, toma en cuenta que el uso de una lista de comprensión es una forma compacta de obtener (comprimir) lo que usualmente se puede hacer con un ciclo FOR:

Usando ciclo FOR:

```
U = [1,2,3,4,5]
P = []

for n in U:
  if n>2:
    P.append(n**2)

print(P)
```

```
[9, 16, 25]
```

Observa que en este caso necesariamente se debe inicializar P.

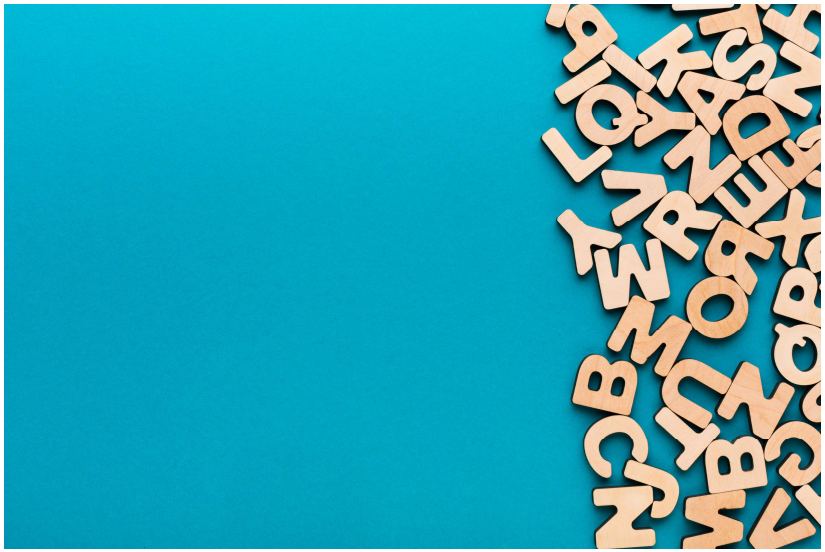
Usando lista de comprensión:

```
U = [1,2,3,4,5]

P = [n**2 for n in U if n>2]

print(P)
```

```
[9, 16, 25]
```

En este caso, la inicialización de P se realiza de manera automática dentro de la misma lista de comprensión.

Procesamiento de texto mediante Expresiones Regulares : regex



A continuación estudiaremos el tema de expresiones regulares, el cual usualmente se abrevia como “regex” o “regexp”, por sus siglas en inglés.

Veremos que las técnicas de regex son muy poderosas para la manipulación de texto en los documentos y por lo tanto en la etapa de pre-procesamiento y tokenización.

En los comentarios de usuarios en redes sociales sobre algún producto adquirido o servicio recibido, es común encontrar frases como las siguientes:

- pésimo servicio
- pesimo serbicio
- PESIMO SERVICIO
- P-É-S-I-M-O S-E-R-V-I-C-I-O
- PEEEEEESIIIMOOOOOOOOOOO servicio
- ¡¡¡¡Peeeeeesimoooo Seeerviiiicioooooo!!!!!!!!!!!!!!!

El objetivo ahora es que a través de un lenguaje formal se pueda entender que todos estos comentarios son sobre una misma idea en cuanto a la calidad del servicio recibido se refiere y que todos ellos serían equivalentes al primero de los enunciados de esta lista.

Para el tratamiento de estos enunciados nos apoyaremos en un lenguaje formal conocido como "**expresiones regulares**" y que se abrevia simplemente como "**regex**" o "**regexp**", por su nombre en inglés "regular expression".

Expresiones Regulares
regex / regexp

Una expresión regular es una fórmula que nos permite especificar cadenas de caracteres.

Una expresión regular está escrita en un lenguaje formal y podrá ser interpretado por un procesador de expresiones comunes.

Así, una expresión regular nos permitirá encontrar cadenas de texto o secuencias de caracteres en un lenguaje formal.

Para llevar a cabo un búsqueda mediante una expresión regular, se requiere de un **patrón** y un **corpus** donde se realice la búsqueda.

En general, trabajaremos con corpus formados por documentos de texto y cada texto estará formado por cadena de caracteres alfanuméricos y caracteres especiales.

Existen una gran cantidad de reglas regex, veremos en esta semana algunas de las principales reglas, pero igualmente a lo largo del curso estaremos introduciendo algunas más cuando sea necesario.

Existen una gran variedad librerías para manejar expresiones regulares, en particular con Python usaremos la llamada “re”, la cual es una de las más completas y bien documentadas:

https://docs.python.org/3/library/re.html

```
import re
```

En el JupyteNotebook de esta semana podrás encontrar una mayor variedad de ejemplos, pero por el momento en esta presentación mostraremos algunos casos para ayudar a la comprensión de cada expresión regular.

Expresiones Regulares : Disyunciones y Negaciones

Operador / Patrón	Comportamiento /Coincidencia
[A-Z]	Cualquier letra mayúscula
[a-z]	Cualquier letra minúscula
[0-9]	Cualquier dígito
[^A-Z]	Cualquier carácter que no sea una letra mayúscula
[^A-Za-z0-9]	Cualquier carácter no alfanumérico

El carácter "^" indica negación solamente cuando es el primer carácter dentro de las corcheas [].

NOTA: En particular
[A-Z] es equivalente a [ABCDEFGHIJKLMNOPQRSTUVWXYZ]
[0-9] es equivalente a [0123456789]

método **findall()** de regex

patrón string

```
import re

re.findall('[A-Z]', 'Hola Mundo')
```

```
['H', 'M']
```

Busca todas las coincidencias en "string" del "patrón" indicado.

Busca todas las coincidencias del string que sea una letra mayúscula.

Puedes agregar otros caracteres en el patrón de búsqueda, por ejemplo:

```
import re

print(re.findall('[a-z]', 'pingüino español'))
print(re.findall('[a-zñü]', 'pingüino español'))
```

```
['p', 'i', 'n', 'g', 'i', 'n', 'o', 'e', 's', 'p', 'a', 'o', 'l']
['p', 'i', 'n', 'g', 'ü', 'i', 'n', 'o', 'e', 's', 'p', 'a', 'ñ', 'o', 'l']
```

Puede también usarse el operador disyunción “|” como sigue:

```
import re

print(re.findall('España|Espana',
                 'El nombre de España a veces se escribe Espana'))
```

```
['España', 'Espana']
```

Busca todo lo que no sea una letra mayúscula, inclusive el espacio en blanco en este caso:

```
print(re.findall('[^A-Z]', 'Hola Mundo'))
```

```
['o', 'l', 'a', ' ', 'u', 'n', 'd', 'o']
```

Si al operador disyunción “|” se le antepone la diagonal invertida “\” en el patrón a buscar, entonces queda desactivada su función de operador y se convierte en otro caracter más:

```
print(re.findall('gato|gata', 'Es un gato|gata'))
print(re.findall('gato\|gata', 'Es un gato|gata'))
```

```
['gato', 'gata']
['gato|gata']
```

Observa que ahora el patrón a buscar es uno solo:

Si el carácter “^” está dentro de las corcheas “[ ]” en el patrón de búsqueda, pero no al inicio de las corcheas, entonces es otro simple carácter más:

```
print(re.findall('[^0-9]', 'año-8-x^2'))
print(re.findall('[0-9^]', 'año-8-x^2'))
```

```
['a', 'ñ', 'o', '-', '-', 'x', '^']
['8', '^', '2']
```

repeticiones: ? * +

Tenemos también los siguientes operadores que nos indican si algún carácter o patrón aparece cierto número de veces

```
print(re.findall('ho?la', 'hla hola hoooola hhhooolaaaa'))
print(re.findall('ho*la', 'hla hola hoooola hhhooolaaaa'))
print(re.findall('ho+la', 'hla hola hoooola hhhooolaaaa'))
```

```
['hla', 'hola']
['hla', 'hola', 'hoooola', 'hooola']
['hola', 'hoooola', 'hooola']
```

El operador "?" nos indica que el carácter previo puede o no aparecer.

El operador de Kleene "*" nos indica que el carácter previo aparece 0, 1, 2, … veces, es decir, puede no aparecer, o bien aparecer varias veces repetido.

El operador de Kleene "+" nos indica que el carácter previo aparece 1, 2, … veces, es decir, debe aparecer al menos una vez.

El nombre se debe al matemático de EEUU Stephen Cole Kleene (1909-1994).

Más operadores regex : ^ $.

El operador “ . ” es un comodín e indica que puede ser cualquier carácter.

El operador ^ indica un patrón al inicio del string.

El operador $ indica un patrón al final del string.

```
print(re.findall('L...',  'Lupa Loro La Luna'))
print(re.findall('^L...', 'Lupa Loro La Luna'))
print(re.findall('...a$', 'Lupa Loro La Luna'))
```

```
['Lupa', 'Loro', 'La L']
['Lupa']
['Luna']
```

A los operadores “^” y “$” también se les llaman anclas (**anchors** en inglés).

NOTA: De lo estudiado previamente, vemos que el operador “^” tiene tres usos: búsqueda de coincidencias al inicio de un string; negación dentro y al inicio de corcheas; simplemente el carácter como tal.

Símbolos especiales

Los siguientes símbolos nos permiten generar búsquedas regex de manera más rápida:

Símbolo	Comportamiento /Coincidencia
\d	Cualquier dígito. Equivalente a [0-9]
\w	Cualquier carácter alfanumérico. Equivalente a [a-zA-Z0-9]
\s	Cualquier generador de espacio(s) en blanco: espacio en blanco, salto de línea (\n), tabulador (\t), etc.
\D	Cualquier carácter que no sea un dígito. Equivalente a [^0-9]
\W	Cualquier carácter no alfanumérico. Equivalente a [^a-zA-Z0-9]
\S	Cualquier carácter que no sea espacio en blanco.

Conflicto entre símbolos regex y los documentos de texto

Observa que todos los operadores y símbolos vistos hasta ahora tienen un significado específico dentro del área de expresiones regulares (regex). Sin embargo, sabemos que todos estos símbolos tienen a su vez su propio significado dentro de los documentos de texto que se estarán analizando, y esto puede generar un conflicto durante el análisis con regex sobre cómo interpretarlos.

Por ejemplo, el símbolo $ ¿deberá interpretarse como símbolo de pesos o como parte de un patrón regex?

Análogamente sabemos que la diagonal invertida seguida de ciertos caracteres tienen diversos significados en los documentos de texto (como los tabuladores, saltos de línea, etc.) y al mismo tiempo se usa para dar significado a patrones en regex (como \w, \W, etc.).

Para evitar estos conflictos, al usar regex, se debe tomar siempre como política general el tratar los strings como textos planos (**raw text** en inglés), para quitar el significado que tengan dentro de los documentos de texto y pueden utilizarse los operadores y símbolos regex sin problemas de conflicto. Esto se logra anteponiendo la letra **r** al string en cuestión. En muchas ocasiones será equivalente usar o no dicha letra **r**, pero cuando se analizan grandes documentos de texto es difícil estar al tanto de si habrá o no conflicto con algunos de estos símbolos o caracteres, pero ello es mejor siempre utilizarlo de aquí en adelante.

Así, en lo sucesivo, escribiremos por ejemplo:

```
re.findall(r'\w+', r'casa, libro--- #tec')
```

en lugar de:

```
re.findall('\w+', 'casa, libro--- #tec')
```

Por ejemplo, observa la diferencia en las siguientes dos salidas.

Si introducimos el siguiente string sin la indicación de texto plano, entonces el símbolo “\n” se interpreta como salto de línea y al imprimirlo dicho símbolo se aplica como tal y obtenemos la siguiente salida:

```
print('Lupa\nLoro\nLa\nLuna')
```

```
Lupa
Loro
La
Luna
```

Sin embargo, si anteponemos la **r** para indicar que queremos interpretar el string como texto plano, se “desactiva” el significado de salto de línea de “\n” y se consideran simplemente como otros dos caracteres consecutivos más:

```
print(r'Lupa\nLoro\nLa\nLuna')
```

```
Lupa\nLoro\nLa\nLuna
```

Así, ya como texto plano, los métodos de regex podrán asignar los significados deseados a todos los símbolos que acabamos de definir, sin preocuparse de que se vaya a generar algún tipo de conflicto.

regex adicionales

Veamos con ejemplos algunos otros casos que llegan a ser muy prácticos en ocasiones:

```
# En todos los casos siguientes se indica entre llaves el número
# de repeticiones que se busca del caracter previo:

print(re.findall(r'ho{3}la', r'hla hola hoola hooola'))    # 3 repeticiones exactas.
print(re.findall(r'ho{2,3}la', r'hla hola hoola hooola'))  # de 2 a 3 repeticiones.
print(re.findall(r'ho{,2}la', r'hla hola hoola hooola'))   # cuando mucho 2 repeticiones.
print(re.findall(r'ho{2,}la', r'hla hola hoola hooola'))   # cuando menos 2 repeticiones.
```

```
['hooola']
['hoola', 'hooola']
['hla', 'hola', 'hoola']
['hoola', 'hooola']
```

método **sub()** de regex

patrónA
patrónB
string

```
import re

re.sub('ñ', 'gn', 'Picasso nació en España.')
```

```
'Picasso nació en Espagna.'
```

Sustituye “patrónA” con “patrónB” en el “string”.

Sustituye la letra “ñ” por las letras “gn” en la frase “string”.

Veamos un ejemplo:

```
re.sub('o+', 'o', 'Hoooola.')
```

```
'Hola.'
```

Paréntesis ( )

Los paréntesis redondos ( ) introducen prioridad de búsqueda y por lo tanto se inicia buscando el patrón dentro de ellos, pero que también coincida con lo que se tenga afuera de dichos paréntesis. Una vez encontrado todo el patrón, solamente extrae lo que está dentro de los paréntesis:

```
re.findall(r"\w+(ing|ed)", r"playing cat bingo loved singing")
```

```
['ing', 'ing', 'ed', 'ing']
```

```
re.findall(r"\w+(?:ing|ed)", r"playing cat bingo loved singing")
```

```
['playing', 'bing', 'loved', 'singing']
```

Los caracteres “?:” dentro de los paréntesis redondos desactivan la función inicial que tienen los paréntesis redondos, a saber, la de extraer solo los caracteres que están dentro de ellos y que hacen match con todo lo indicado en la regex. Por ello, en este último ejemplo se regresa la coincidencia de todos los caracteres indicados en el patrón y en el primero que no se usan estos símbolos, solo se regresa lo que está dentro de los paréntesis redondos.

Frontera o Límite de palabras (Word boundary) \b

“\b” busca cualquier carácter que sea límite entre un “word character” y un “no word character”.

Recordemos que un “word character” en lenguajes de programación representa una letra, un dígito o el guion bajo (underscore).

```
re.findall(r"\w+(?:ing|ed)\b", r"playing cat bingo loved singing")
```

```
['playing', 'loved', 'singing']
```

Entre otros caracteres, un “no word character” podría ser el inicio o final del string, o un espacio en blanco.

Ejemplos Opcionales: poco a poco irás asimilando cada uno de estos símbolos, pero aquí te damos otros ejemplos. En dado caso puedes regresar más adelante a estudiar estos ejemplos que siguen. Para una primera lectura puedes omitir el resto de la diapositivas.

Analiza bien estos ejemplos, pero recuerda que con "\b" básicamente se está buscando un "no-word-char" seguido de un "word-char", o al revés.

```
re.findall(r'\bE\d', r'E1 E2 #E3 _E4 3E5 aE6 aE7$E8')
```

```
['E1', 'E2', 'E3', 'E8']
```

Estamos buscando "no-word-chars" que antecedan a la letra "E" seguida de un dígito. Las "no-word-chars" encontradas son el inicio del string, el espacio en blanco, el # y el $.

También podemos usar la negación de "\b", que se denota "\B":

```
re.findall(r'\BE\d', r'E1 E2 #E3 _E4 3E5 aE6 aE7$E8')
```

```
['E4', 'E5', 'E6', 'E7']
```

Ahora estamos buscando "word-chars" que antecedan a la letra "E" seguida de un dígito. Las "word-chars" encontradas son el guion bajo, el 3 y la letra "a" dos veces.

Ejemplo Opcional

Se pueden usar paréntesis anidados, la manera en que se aplican (enumeran) es de izquierda a derecha, como se ilustra en el siguiente ejemplo:

```
fechas = r'19-02-1978 22-11-2015 30/09/2021'

print(re.findall(r'\d{2}-\d{2}-\d{4}', fechas))
print(re.findall(r'((\d{2}-\d{2})-(\d{4}))', fechas))
print(re.findall(r'((\d{2}[-/]\d{2})[-/](\d{4}))', fechas))
```

```
['19-02-1978', '22-11-2015']
[('19-02-1978', '19-02', '1978'), ('22-11-2015', '22-11', '2015')]
[('19-02-1978', '19-02', '1978'), ('22-11-2015', '22-11', '2015'), ('30/09/2021', '30/09', '2021')]
```

En el primer regex del primer paréntesis, se buscan directamente todas las fechas que están en el formato DD-MM-YYYY.

En el segundo regex se busca el patrón de fechas en el formato DD-MM-YYYY, seguido de DD-MM (estrictamente del patrón DD-MM-YYYY extrae los dos dígitos, seguidos de un guion y de otros dos dígitos) y finalmente de este mismo patrón general extrae el año (estrictamente los cuatro últimos números seguidos del patrón).

El tercer regex solo incluye fechas en formato usando la diagonal, en lugar del guion.

Ejemplo Opcional

```
import re

X = "To say 'I don't have time', is like saying, 'I don't want to'. -Lao-Tzu"
print(X)
```

```
To say 'I don't have time', is like saying, 'I don't want to'. -Lao-Tzu
```

```
re.findall(r"\w*(?:['-]\w+)", X)
```

```
["'I", "don't", "'I", "don't", '-Lao', '-Tzu']
```

Busca todas las secuencias de strings compuestas por una o más letras o números (de forma opcional), seguidas por un apóstrofe o guion y nuevamente seguido de letras o Números.

Recuerda que los caracteres “?:” desactivan la función inicial que tienen los paréntesis redondos, extraer solo los caracteres que están dentro de ellos y que hacen match con todo lo indicado en la regex.

Ejemplo Opcional

Los paréntesis redondos ( ) van guardando la cadena de strings que cumplen el patrón de que se expresa dentro de ellos. Y lo van haciendo de manera jerarquizada, de izquierda a derecha. Se numeran con el formato "\n" donde "n" es el "n-ésimo" paréntesis redondo que se aplicó.

Esto lo puedes utilizar por ejemplo para sustituir patrones de caracteres.

Veamos algunos ejemplos:

```
docs = ['Me siento nervioso', 'Me siento cansado', 'Me siento triste']

for doc in docs:
  print(re.sub(r'Me siento (.+)', r'¿Por qué te sientes \1?', doc))
```

```
¿Por qué te sientes nervioso?
¿Por qué te sientes cansado?
¿Por qué te sientes triste?
```

Con "(.+)" se busca cualquier cadena de caracteres después del espacio en blanco de la palabra "siento" y eso lo guarda en "\1", el cual puedes mandar llamar después, como en este caso al hacer la sustitución.

Tecnológico de Monterrey

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