



## Unidad 3 - Procesamiento del Lenguaje

### 1. Extracción de frases sustantivas (Noun phrase extraction)

En el procesamiento del lenguaje natural (NLP), la extracción de frases sustantivas ("Extracting Noun Phrases") se refiere al proceso de identificar y extraer frases que funcionan como sustantivos en un texto.

Una frase sustantiva, también conocida como sintagma nominal, es una estructura gramatical que tiene un sustantivo como su núcleo o palabra principal. La función principal de una frase sustantiva es nombrar o identificar personas, lugares, cosas, ideas o eventos. Puede actuar como sujeto, objeto directo, objeto indirecto, complemento de preposición, entre otros, dentro de una oración.

#### Estructura de una Frase Sustantiva

Una frase sustantiva puede estar compuesta por:

1. **Núcleo:** Es el sustantivo principal de la frase y es el elemento obligatorio en la frase sustantiva.
2. **Determinante:** Palabras como artículos, posesivos, demostrativos, etc., que acompañan al sustantivo y aportan información sobre él.
3. **Modificadores:** Adjetivos, frases adjetivas, o frases preposicionales que describen o dan más información sobre el sustantivo.
4. **Complementos:** Palabras o frases que completan el sentido del sustantivo.

#### Ejemplos de Frases Sustantivas

##### 1. El perro grande (Determinante + Núcleo + Modificador)

- "El" es el determinante.
- "perro" es el núcleo.
- "grande" es el modificador.

##### 2. La casa de Juan (Determinante + Núcleo + Complemento)

- "La" es el determinante.
- "casa" es el núcleo.
- "de Juan" es el complemento.

##### 3. Un libro interesante (Determinante + Núcleo + Modificador)

- "Un" es el determinante.
- "libro" es el núcleo.
- "interesante" es el modificador.

Veamos un ejemplo usando `spacy`, que cuenta con modelos para idioma español. Primero instalamos los paquetes necesarios:

```
pip install spacy  
python -m spacy download es_core_news_lg
```

Luego podemos correr el siguiente ejemplo:

```
import spacy  
  
# Carga el modelo de lenguaje preentrenado de Spacy  
nlp = spacy.load('es_core_news_lg')  
  
# Procesa una oración con el modelo de Spacy  
doc = nlp("El veloz zorro marrón salta sobre el perro perezoso.")  
  
# Extrae e imprime las frases nominales  
for chunk in doc.noun_chunks:  
    print(chunk.text)
```

Y el resultado será:

```
El veloz zorro marrón  
el perro perezoso
```

En la oración "El veloz zorro marrón salta sobre el perro perezoso", se pueden identificar dos frases sustantivas:

#### 1. El veloz zorro marrón

- "El" es el determinante.
- "zorro" es el núcleo o sustantivo principal de la frase sustantiva.
- "veloz" y "marrón" son modificadores, específicamente adjetivos que describen al zorro.

#### 2. El perro perezoso

- "El" es el determinante.
- "perro" es el núcleo o sustantivo principal de la frase sustantiva.
- "perezoso" es un modificador, específicamente un adjetivo que describe al perro.

## 2. Semejanza de texto (Text similarity)

La semejanza de texto, también conocida como "Text Similarity" en inglés, es una área de estudio en el procesamiento del lenguaje natural (NLP) que se centra en determinar el grado de similitud o equivalencia entre dos fragmentos de texto. Este concepto es fundamental para varias aplicaciones de NLP, como la búsqueda semántica, la agrupación de documentos, la detección de plagio, entre otros. Veamos cuáles son los aspectos clave sobre la semejanza de texto:

### Métricas de Semejanza

Existen varias métricas y técnicas para calcular la semejanza de texto, incluyendo:

1. **Similitud del coseno**: Utiliza el ángulo entre dos vectores en un espacio vectorial para determinar la similitud entre ellos. Es ampliamente utilizado con técnicas de vectorización de texto como TF-IDF y embeddings de palabras.
2. **Distancia de Jaccard**: Calcula la semejanza entre dos conjuntos, generalmente se utiliza para comparar conjuntos de palabras o caracteres.
3. **Distancia de Levenshtein** (o distancia de edición): Mide el número mínimo de operaciones (inserciones, eliminaciones o sustituciones) requeridas para transformar una cadena de caracteres en otra.
4. **Similitud de Dice**: Es una métrica de semejanza que relaciona dos veces el número de elementos comunes dividido por el número total de elementos.

### Aplicaciones

La semejanza de texto tiene aplicaciones en una variedad de campos, incluyendo:

1. **Sistemas de Recomendación**: Para recomendar contenido similar al que un usuario ha interactuado anteriormente.
2. **Detección de Plagio**: Para identificar casos de copia o plagio de texto.

3. **Respuesta Automática a Preguntas (QA)**: Para encontrar la mejor respuesta a una pregunta dada en una base de datos de conocimientos.
4. **Clasificación de Textos**: Para agrupar textos similares en las mismas categorías.

### Similitud del coseno

Veamos un ejemplo concreto. Aquí usaremos la codificación `TfidfVectorizer`, y **Similitud del coseno** que hemos visto en la unidad anterior :

```
# Importar bibliotecas
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Lista de documentos (frases en español)
documents = [
    "El cielo es azul y despejado hoy.",
    "Me gusta salir a caminar bajo el cielo azul.",
    "Hoy el clima está muy agradable.",
    "Disfruto de un día soleado con el cielo despejado.",
    "Los días soleados me hacen sentir feliz."
]

# Calcular TF-IDF: ingeniería de características
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)

# Mostrar las dimensiones de la matriz TF-IDF
print(tfidf_matrix.shape)

# Calcular la similitud del coseno para la primera oración con el resto de las oraciones
similarity_matrix = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix)

# Encontrar el índice de la frase más semejante (excluyendo la primera frase)
most_similar_index = similarity_matrix.argsort()[0, -2]

# Imprimir la frase más semejante
print(f"La frase más semejante a '{documents[0]}' es: '{documents[most_similar_index]}'")

# Salida del script:
# (5, 27)
# La frase más semejante a 'El cielo es azul y despejado hoy.' es: 'Me gusta salir a caminar bajo el cielo azul.'
```

Después de calcular la matriz de similitud del coseno, utilizamos `argsort` para obtener los índices de las frases ordenadas por similitud del coseno. Utilizamos `2` para obtener el índice de la frase más semejante que no sea la primera frase (ya que la similitud del coseno de una frase consigo misma será siempre 1).

### Distancia de Coseno:

Es una transformación de la similitud de coseno para representar una idea de "distancia" o "disimilitud". Se calcula como:

`Distancia de Coseno = 1 - Similitud de Coseno`. Su valor oscila entre 0 y 2:

- Si los dos vectores son idénticos, su distancia de coseno es 0.
- Si son completamente opuestos, la distancia es 2.
- Si son ortogonales, la distancia es 1.

Es útil cuando se desea tener una métrica que represente la noción de cuán lejos están dos vectores entre sí.

### Distancia de Jaccard

La Similitud de Jaccard, también conocida como el coeficiente de Jaccard, es una métrica utilizada para comparar la similitud entre dos conjuntos midiendo la Distancia de Jaccard. Es especialmente útil en el campo del Procesamiento del Lenguaje Natural (NLP) para medir la similitud entre dos textos y se calcula utilizando la siguiente fórmula:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Donde:

- $A$  y  $B$  son los dos conjuntos que se están comparando.

- $|A \cap B|$  es el número de elementos en la intersección de  $A$  y  $B$  (elementos comunes entre  $A$  y  $B$ ).
- $|A \cup B|$  es el número de elementos en la unión de  $A$  y  $B$  (elementos que aparecen en  $A$ , en  $B$ , o en ambos).

El valor de la distancia de Jaccard varía entre 0 y 1, donde 0 indica que los conjuntos son completamente diferentes (disjuntos) y 1 indica que son idénticos.

En NLP, los conjuntos suelen ser conjuntos de palabras (tokens) o n-gramas extraídos de documentos o textos. Por ejemplo, para comparar las frases "El gato juega con la pelota" y "El perro corre tras la pelota" tendríamos los siguientes conjuntos:

- Conjunto  $A$ : {El, gato, juega, con, la, pelota}
- Conjunto  $B$ : {El, perro, corre, tras, la, pelota}

Podemos calcular la distancia en Python del siguiente modo:

```
def jaccard_similarity(set_a, set_b):
    intersection = set_a.intersection(set_b)
    union = set_a.union(set_b)
    return len(intersection) / len(union)

# Ejemplo de uso:
frase_1 = "El gato juega con la pelota"
frase_2 = "El perro corre tras la pelota"

set_a = set(frase_1.split())
set_b = set(frase_2.split())

similarity = jaccard_similarity(set_a, set_b)
print(f"Jaccard Similarity: {similarity}")

intersection = set_a.intersection(set_b)
union = set_a.union(set_b)
print(f"Intersección: {intersection}")
print(f"Unión: {union}")

# Imprime:
# Jaccard Similarity: 0.3333333333333333
# Intersección: {'pelota.', 'El', 'la'}
# Unión: {'gato', 'la', 'pelota.', 'perro', 'El', 'juega', 'con', 'corre', 'tras'}
```

También podemos utilizar `jaccard_score` de `sklearn` como en el siguiente ejemplo. Con `jaccard_score` será necesario antes convertir el texto a vectores, por ejemplo con `CountVectorizer` o la técnica de One Hot encoding.

```
from sklearn.metrics import jaccard_score
from sklearn.feature_extraction.text import CountVectorizer

frases = [
    ("El gato negro salta sobre el sofá.", "El gato negro salta encima del sofá."),
    ("Los niños juegan en el parque con una pelota.", "Los niños corren en el parque con un balón."),
    ("El sol brilla en el cielo azul durante el día.", "La luna ilumina la noche estrellada."),
    ("El elefante camina lentamente por la selva.", "La computadora procesa rápidamente los datos.")
]

for frase_a, frase_b in frases:
    vectorizer = CountVectorizer(binary=True)
    X = vectorizer.fit_transform([frase_a, frase_b])

    vector_1 = X[0].toarray()[0]
    vector_2 = X[1].toarray()[0]

    similarity = jaccard_score(vector_1, vector_2, average='micro')
    print(f"Jaccard Similarity between\n\"{frase_a}\"\nand\n\"{frase_b}\"\nis: {similarity}\n")
```

Y los resultados serán:

```
Jaccard Similarity between
"El gato negro salta sobre el sofá."
and
"El gato negro salta encima del sofá."
is: 0.45454545454545453

Jaccard Similarity between
"Los niños juegan en el parque con una pelota."
and
"Los niños corren en el parque con un balón."
```

```

is: 0.3333333333333333

Jaccard Similarity between
"El sol brilla en el cielo azul durante el día."
and
"La luna ilumina la noche estrellada."
is: 0.0

Jaccard Similarity between
"El elefante camina lentamente por la selva."
and
"La computadora procesa rápidamente los datos."
is: 0.043478260869565216

```

### Similitud de Dice

La Similitud de Dice, también conocida como coeficiente de Sørensen-Dice, es una métrica utilizada para calcular la similitud entre dos conjuntos. Es especialmente útil en el campo del Procesamiento del Lenguaje Natural (NLP) para medir la similitud entre dos textos o documentos. La similitud de Dice se calcula utilizando la siguiente fórmula:

$$D(A, B) = \frac{2 \cdot |A \cap B|}{|A| + |B|}$$

Donde:

- $A$  y  $B$  son los dos conjuntos que se están comparando.
- $|A \cap B|$  es el número de elementos en la intersección de  $A$  y  $B$  (elementos comunes entre  $A$  y  $B$ ).
- $|A|$  y  $|B|$  son el número de elementos en los conjuntos  $A$  y  $B$  respectivamente.

Podemos implementarlo en Python de este modo:

```

def dice_similarity(set_a, set_b):
    intersection = set_a.intersection(set_b)
    return 2 * len(intersection) / (len(set_a) + len(set_b))

frase_1 = "El gato juega"
frase_2 = "El perro juega"

set_a = set(frase_1.split())
set_b = set(frase_2.split())

similarity = dice_similarity(set_a, set_b)
print(f"Dice Similarity: {similarity}")

# Imprime:
# 0.6666666666666666

```

Este resultado indica una similitud del 66.6% entre las dos frases, ya que comparten algunos tokens, pero también tienen diferencias.



La Similitud de Dice es generalmente más sensible a las coincidencias (elementos en común) que la Similitud de Jaccard, ya que el numerador en la fórmula de Dice se multiplica por 2. Esto significa que, para dos conjuntos con el mismo número de coincidencias y diferencias, la Similitud de Dice generalmente dará un valor más alto que la Similitud de Jaccard.

### Distancia de Levenshtein

La Distancia de Levenshtein, también conocida como distancia de edición, es una métrica que mide cuánto se diferencian dos secuencias de caracteres (por ejemplo, dos cadenas de texto). La distancia entre dos cadenas es el número mínimo de operaciones de edición únicas requeridas para transformar una cadena en la otra. Las operaciones de edición permitidas son:

1. **Inserción:** Agregar un nuevo carácter a la cadena.
2. **Eliminación:** Quitar un carácter de la cadena.

3. **Sustitución**: Cambiar un carácter por otro.

En Python, la Distancia de Levenshtein se puede calcular utilizando la librería [python-Levenshtein](#) (<https://github.com/maxbachmann/python-Levenshtein>):

```
# !pip install python-Levenshtein
import Levenshtein

s = "coseno"
t = "obsceno"

distance = Levenshtein.distance(s, t)
print(f"Levenshtein Distance: {distance}")

# Imprime
# Levenshtein Distance: 3
```

## Coincidencia fonética (Phonetic Matching)

El "Phonetic Matching" o coincidencia fonética es una técnica utilizada para encontrar palabras que suenan de manera similar, aunque pueden tener diferentes ortografías. En el contexto del procesamiento del lenguaje natural (NLP), se utiliza para varias aplicaciones como la corrección ortográfica, la búsqueda de palabras clave, la desambiguación de entidades, entre otros. El objetivo principal es identificar palabras que tienen una pronunciación similar. Esto es útil, por ejemplo, en la búsqueda de nombres propios donde las personas pueden tener varias formas de escribir un nombre que suena igual o similar, o para corregir transcripción de texto.

Para ver un ejemplo, podríamos usar la librería [pyphonetics](#) (<https://github.com/Lilykos/pyphonetics>)

```
# Instalamos la librería que vamos a utilizar:
# !pip install pyphonetics
from pyphonetics import Soundex

# Crear una instancia del objeto Soundex
soundex = Soundex()

# Palabras para comparar
word1 = 'two'
word2 = 'too'

# Obtener las representaciones Soundex de las palabras
soundex1 = soundex.phonetics(word1)
soundex2 = soundex.phonetics(word2)

# Comparar las representaciones Soundex
if soundex1 == soundex2:
    print(f"Las palabras '{word1}' y '{word2}' son fonéticamente similares según Soundex.")
else:
    print(f"Las palabras '{word1}' y '{word2}' no son fonéticamente similares según Soundex.")
```

Y el resultado será:

```
Las palabras 'two' y 'too' son fonéticamente similares según Soundex.
```

Aunque hemos utilizado el algoritmo Soundex en este ejemplo, [pyphonetics](#) también ofrece otros algoritmos de coincidencia fonética, como Refined Soundex, Metaphone, y más. Podemos experimentar con estos otros algoritmos cambiando la clase que importamos (por ejemplo, `from pyphonetics import Metaphone` y `metaphone = Metaphone()`).

También podemos calcular la distancia fonética entre dos pares de palabras utilizando el algoritmo Refined Soundex y dos métricas diferentes para calcular la distancia. Aquí está el desglose del código:

```
from pyphonetics import RefinedSoundex

# Crear una instancia del objeto RefinedSoundex
rs = RefinedSoundex()

# Calcular y imprimir la distancia fonética entre 'Mister' y 'Master'
# usando la métrica predeterminada (Levenshtein)
dist = rs.distance('Mister', 'Master')
```

```

print(dist)    # Imprime 0

# Calcular y imprimir la distancia fonética entre 'assign' y 'assist'
# usando la métrica de Hamming
dist = rs.distance('assign', 'assist', metric='hamming')
print(dist)    # Imprime 2

```

La distancia cero en este contexto indica que, según el algoritmo Refined Soundex, las palabras "Mister" y "Master" tienen representaciones fonéticas idénticas. Esto no significa que las palabras sean idénticas, sino que suenan muy similares fonéticamente.

El algoritmo Soundex y su variante, el Refined Soundex, funcionan codificando palabras en una serie de caracteres que representan grupos de consonantes fonéticamente similares. Las vocales no se codifican, y solo se considera la primera letra de la palabra. Esto significa que las palabras que comienzan con la misma letra y tienen una estructura consonántica similar tendrán el mismo código Soundex o Refined Soundex.

En el caso de "Mister" y "Master", ambas palabras comienzan con la letra "M" y luego tienen una consonante "s" (o "st" que se codifica como una única unidad en algunos sistemas Soundex) seguida de una "t". Esto lleva a que tengan el mismo código Refined Soundex, y por lo tanto una distancia de 0 cuando se comparan usando una métrica de distancia como la de Levenshtein.

En el otro caso, se utiliza la **distancia de Hamming**, que es una métrica utilizada para medir la diferencia entre dos cadenas de **igual longitud**. Específicamente, representa el número de posiciones en las que los correspondientes elementos son diferentes.

### 3. POS (Parts of speech tagging) y NER (Named Entity Recognition)

#### NER

El término "NER" se refiere a "Reconocimiento de Entidades Nombradas" (Named Entity Recognition en inglés). Es un subprocesso del Procesamiento del Lenguaje Natural (NLP) que se centra en identificar y clasificar entidades nombradas presentes en un texto en categorías predefinidas como nombres de personas, organizaciones, lugares, expresiones de tiempo, cantidades, valores monetarios, porcentajes, etc.

Formalmente el concepto de «entidad nombrada» se deriva de la definición del filósofo estadounidense Saul Kripke de designador rígido (Kripke, 1980) que forma parte de la lógica modal y filosofía del lenguaje.

#### Características y Funcionamiento

1. **Identificación de Entidades:** Localiza y delimita las entidades nombradas en un texto.
2. **Clasificación de Entidades:** Una vez identificadas las entidades, las clasifica en diversas categorías como persona, organización, lugar, etc.
3. **Contexto:** Utiliza el contexto y la estructura gramatical del texto para identificar y clasificar correctamente las entidades.

#### Aplicaciones

El NER tiene una amplia gama de aplicaciones, incluyendo:

- **Sistemas de Recomendación:** Para entender y analizar las preferencias del usuario basándose en las entidades mencionadas en los textos que consume.
- **Búsqueda Semántica:** Para mejorar los motores de búsqueda identificando entidades específicas en los documentos y relacionándolas con las consultas de búsqueda.
- **Análisis de Sentimientos:** Para identificar entidades específicas mencionadas en los textos y analizar los sentimientos asociados con ellas.

Existen diversos modelos capaces de realizar NER, con diferente efectividad. El progreso histórico podemos verlo en la siguiente página: <https://paperswithcode.com/sota/named-entity-recognition-ner-on-conll-2003>, donde podemos visualizar la precisión y el estado del arte de estos modelos (SOTA - State Of The Art).

Veamos un ejemplo de como realizar NER en Python, usando spaCy:

```

#!/usr/bin/env python3
# !pip install spacy
# !python -m spacy download es_core_news_lg

import spacy

```

```

# Cargar el modelo de lenguaje preentrenado en español
nlp = spacy.load('es_core_news_lg')

#Texto a analizar
texto = """Antonio Guterres indicó que el clima está implosionando más rápido de lo que podemos hacer frente. Autoridades europeas indicaron que el verano boreal de 2023 fue el más cálido desde que se tiene registro. Las temperaturas medias mundiales durante los tres meses del verano boreal, fueron las más elevadas desde que se tiene registro, anunció. Nuestro clima está implosionando más rápido de lo que podemos hacer frente, con fenómenos meteorológicos extremos que afectan a todos los continentes. Canículas, sequías, inundaciones o incendios azotaron durante ese verano boreal Asia, Europa y América del Norte, en proporciones dramáticas.

# Procesar el texto con el modelo de spaCy
doc = nlp(texto)

# Imprimir las entidades nombradas, etiquetas y explicaciones
for ent in doc.ents:
    print(f'Entidad: {ent.text}, Etiqueta: {ent.label_}, Explicación: {spacy.explain(ent.label_)}')

```

Y el resultado será:

```

Entidad: Antonio Guterres, Etiqueta: PER, Explicación: Named person or family.
Entidad: Copernicus, Etiqueta: LOC, Explicación: Non-GPE locations, mountain ranges, bodies of water
Entidad: Nuestro, Etiqueta: PER, Explicación: Named person or family.
Entidad: Canículas, Etiqueta: LOC, Explicación: Non-GPE locations, mountain ranges, bodies of water
Entidad: Asia, Etiqueta: LOC, Explicación: Non-GPE locations, mountain ranges, bodies of water
Entidad: Europa, Etiqueta: LOC, Explicación: Non-GPE locations, mountain ranges, bodies of water
Entidad: América del Norte, Etiqueta: LOC, Explicación: Non-GPE locations, mountain ranges, bodies of water

```

Vemos como el modelo, detecta las entidades nombradas en el texto suministrado. Es importante mencionar que el modelo en español no es tan robusto como el modelo en inglés, y puede indicar falsas entidades, o bien puede no detectar algunas de ellas. El procesador de `spacy` permite detectar las siguientes entidades:

```

PERSON: Personas, incluyendo ficticias.
NORP: Nacionalidades o grupos religiosos o políticos.
FAC: Edificios, aeropuertos, autopistas, puentes, etc.
ORG: Empresas, agencias, instituciones, etc.
GPE: Paises, ciudades, estados.
LOC: Ubicaciones que no son GPE, cordilleras, cuerpos de agua.
PRODUCT: Objetos, vehículos, alimentos, etc. (No servicios.)
EVENT: Huracanes nombrados, batallas, guerras, eventos deportivos, etc.
WORK_OF_ART: Títulos de libros, canciones, etc.
LAW: Documentos nombrados convertidos en leyes.
LANGUAGE: Cualquier idioma nombrado.
DATE: Fechas o períodos absolutos o relativos.
TIME: Tiempos menores a un día.
PERCENT: Porcentaje, incluyendo "%".
MONEY: Valores monetarios, incluyendo unidad.
QUANTITY: Mediciones, como de peso o distancia.
ORDINAL: "primero", "segundo", etc.
CARDINAL: Numerales que no entran en otro tipo.

```

Podemos aprovechar el visualizador `displacy` que incluye la librería `spacy`, para ver los resultados de forma más atractiva en Colab:

```

#Visualizador incluido en spacy
from spacy import displacy

for sent in doc.sents:
    displacy.render(nlp(sent.text), style='ent', jupyter=True)

```

Resultado:

□ Antonio Guterres PER indicó que el clima está implosionando más rápido de lo que podemos hacer frente. Autoridades europeas indicaron que el verano boreal de 2023 fue el más cálido desde que se tiene registro. Las temperaturas medias mundiales durante los tres meses del verano boreal, fueron las más elevadas desde que se tiene registro, anunció este miércoles el observatorio europeo Copernicus LOC, para el que 2023 será probablemente el año más caluroso de la historia. Nuestro clima está implosionando más rápido de lo que podemos hacer frente, con fenómenos meteorológicos extremos que afectan a todos los rincones del planeta, alertó en un comunicado, recordando que los científicos llevan mucho tiempo advirtiendo de las consecuencias de nuestra dependencia de los combustibles fósiles. Canículas LOC, sequías, inundaciones o incendios azotaron durante ese verano boreal Asia LOC, Europa LOC y América del Norte LOC, en proporciones dramáticas y a veces inéditas, con pérdidas de vidas humanas y grandes daños en las economías y el medioambiente.

## POS

La etiquetación de partes del discurso (POS, por sus siglas en inglés) es otra parte crucial del procesamiento del lenguaje natural que involucra etiquetar las palabras con una parte del discurso, como sustantivo, verbo, adjetivo, etc. El POS es la base para la resolución de NER, respuesta a preguntas y desambiguación del sentido de las palabras.

### Características y Funcionamiento

1. **Granularidad:** Puede variar desde una categorización básica (como sustantivos, verbos, adjetivos, etc.) hasta categorías más detalladas que incluyen género, número, tiempo, etc.
2. **Dependencia del Contexto:** La categorización de una palabra puede depender del contexto en el que se encuentra.
3. **Riqueza Lingüística:** Ayuda a entender las complejidades lingüísticas del texto, proporcionando una rica anotación lingüística.
4. **Análisis Morfosintáctico:** Identifica la raíz de las palabras y sus afijos para determinar la parte del discurso.
5. **Desambiguación:** Utiliza el contexto para desambiguar palabras que pueden tener más de una categoría gramatical.
6. **Reglas Gramaticales y Estadísticas:** Utiliza reglas gramaticales predefinidas y modelos estadísticos para etiquetar las palabras correctamente.

### Aplicaciones

1. **Desambiguación del Sentido de las Palabras (WSD):** El etiquetado POS es fundamental para los sistemas WSD, que identifican el sentido correcto de una palabra en un contexto específico.
2. **Reconocimiento de Entidades Nombradas (NER):** El etiquetado POS puede ser un paso previo al NER, ayudando a identificar sustantivos propios y otras entidades importantes en un texto.
3. **Ánalisis Sintáctico:** El etiquetado POS es un paso esencial en el análisis sintáctico, que involucra la construcción de árboles sintácticos que representan la estructura gramatical de las oraciones.
4. **Traducción Automática:** Los sistemas de traducción automática utilizan el etiquetado POS para entender la estructura gramatical de las oraciones y traducirlas correctamente.
5. **Respuesta a Preguntas:** Los sistemas de respuesta a preguntas utilizan el etiquetado POS para entender las preguntas formuladas por los usuarios y encontrar respuestas precisas.
6. **Ánalisis de Sentimientos:** El etiquetado POS puede ayudar a identificar adjetivos y adverbios que a menudo llevan una fuerte carga emocional, facilitando el análisis de sentimientos.

Veamos un ejemplo con spacy, para realizar POS de un texto en español:

```
#!pip install spacy
#!python -m spacy download es_core_news_lg

import spacy
import pandas as pd

# Cargar el modelo preentrenado para español
nlp = spacy.load('es_core_news_lg')

# Texto de ejemplo en español
texto = "Juan y Pedro fueron al parque a jugar con sus amigos, mientras el perro buscaba un hueso."

# Procesar el texto con el modelo de spaCy
doc = nlp(texto)

# Crear una lista para almacenar las palabras, etiquetas POS y explicaciones
data = []
```

```

# Iterar sobre los tokens en el Doc y agregar los detalles a la lista de datos
for token in doc:
    data.append([token.text, token.pos_, spacy.explain(token.pos_)])

# Crear un DataFrame a partir de la lista de datos
df = pd.DataFrame(data, columns=['Palabra', 'Etiqueta POS', 'Explicación'])

# Imprimir el DataFrame
print(df)

```

Y obtendremos una tabla como la siguiente:

	Palabra	Etiqueta POS	Explicación
0	Juan	PROPN	proper noun
1	y	CCONJ	coordinating conjunction
2	Pedro	PROPN	proper noun
3	fueron	AUX	auxiliary
4	al	ADP	adposition
5	parque	NOUN	noun
6	a	ADP	adposition
7	jugar	VERB	verb
8	con	ADP	adposition
9	sus	DET	determiner
10	amigos	NOUN	noun
11	,	PUNCT	punctuation
12	mientras	CCONJ	coordinating conjunction
13	el	DET	determiner
14	perro	PROPN	proper noun
15	buscaba	VERB	verb
16	un	DET	determiner
17	hueso	NOUN	noun
18	.	PUNCT	punctuation

También podemos visualizar un gráfico con la estructura del texto etiquetado, usando `displacy`:

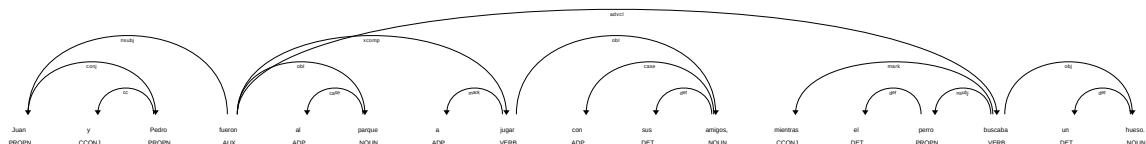
```

from spacy import displacy

# Muestra la gráfica en Jupyter o Colab
displacy.render(doc, style='dep', jupyter=True)

# Guarda la imagen en un archivo SVG
from pathlib import Path
svg = displacy.render(doc, style="dep")
output_path = Path("./dependency_plot.svg")
output_path.open("w", encoding="utf-8").write(svg)

```



La siguiente tabla describe las etiquetas POS que podemos identificar con `spacy`:

POS	DESCRIPCIÓN	EJEMPLOS
ADJ	Adjetivo	grande, viejo, verde, incomprendible, primero
ADP	Adposición	en, para, durante
ADV	Adverbio	muy, mañana, abajo, dónde, ahí
AUX	Auxiliar	es, ha (hecho), será (hacer), debería (hacer)
CONJ	Conjunción	y, o, pero
CCONJ	Conjunción coordinante	y, o, pero
DET	Determinante	un, una, el, la
INTJ	Interjección	¡eh!, ¡ay!, ¡bravo!, ¡hola!

POS	DESCRIPCIÓN	EJEMPLOS
NOUN	Sustantivo	chica, gato, árbol, aire, belleza
NUM	Numeral	1, 2017, uno, setenta y siete, IV, MMXIV
PART	Partícula	del, al, se
PRON	Pronombre	yo, tú, él, ella, nosotros, alguien
PROPN	Sustantivo propio	María, Juan, Madrid, OTAN, HBO
PUNCT	Puntuación	., ( ), ?
SCONJ	Conjunción subordinante	si, mientras, que
SYM	Símbolo	\$, %, §, ©, +, -, ×, ÷, =, :, 😊
VERB	Verbo	correr, corre, corriendo, comer, comió, comiendo
X	Otro	sfpkstdpsxmsa
SPACE	Espacio	

Esta tabla incluye las etiquetas POS más comunes que podemos encontrar en un texto.

Otra librería que podemos usar para realizar POS, es `stanza`. Veamos un ejemplo de como hacerlo:

Y el resultado será:

[Sentence 1]			
El	el	DET	2 det
autobot	autobot	NOUN	11 nsubj
de	de	ADP	4 case
aspecto	aspecto	NOUN	2 nmod
humanoide	humanoide	ADJ	4 amod
,	,	PUNCT	7 punct
llamado	llamado	ADJ	2 amod
Apollo	Apollo	PROPN	7 obj
,	,	PUNCT	7 punct
está	estar	AUX	11 cop
pensado	pensado	ADJ	0 root
para	para	ADP	13 mark
evitar	evitar	VERB	11 advcl
que	que	SCONJ	20 mark
el	el	DET	16 det
ser	ser	NOUN	20 nsubj
humano	humano	ADJ	16 amod
tenga	tener	VERB	13 ccomp
que	que	SCONJ	20 cc
afrontar	afrontar	VERB	18 conj
tareas	tarea	NOUN	20 obj
tediosas	tedioso	ADJ	21 amod
y	y	CCONJ	24 cc
agotadoras	agotador	ADJ	22 conj
.	.	PUNCT	11 punct
[Sentence 2]			
El	el	DET	2 det
artefacto	artefacto	NOUN	18 nsubj
de	de	ADP	5 case
la	el	DET	5 det

empresa	empresa	NOUN	2 nmod
de	de	ADP	7 case
tecnología	tecnología	NOUN	5 nmod
Apptronik	Apptronik	PROPN	5 appos
,	,	PUNCT	11 punct
con	con	ADP	11 case
sede	sede	NOUN	5 nmod
en	en	ADP	13 case
Austin	Austin	PROPN	11 nmod
,	,	PUNCT	15 punct
Texas	Texas	PROPN	13 flat
,	,	PUNCT	11 punct
ya	ya	ADV	18 advmod
realiza	realizar	VERB	0 root
sus	su	DET	21 det
primeras	primero	ADJ	21 amod
tareas	tarea	NOUN	18 obj
en	en	ADP	24 case
una	uno	DET	24 det
empresa	empresa	NOUN	18 obl
.	.	PUNCT	18 punct

Como podemos observar, en la tercera columna de nuestro ejemplo, tenemos las etiquetas POS. También, para cada palabra, imprimimos varios detalles:

- `word.text`: El texto de la palabra.
- `word.lemma`: El lema de la palabra, es decir, su forma canónica.
- `word.pos`: La etiqueta de parte del habla (part of speech, POS) de la palabra.
- `word.head`: El índice de la palabra "cabeza" en la relación de dependencia sintáctica de la palabra.
- `word.deprel`: La etiqueta que describe la relación de dependencia sintáctica de la palabra con su palabra "cabeza".

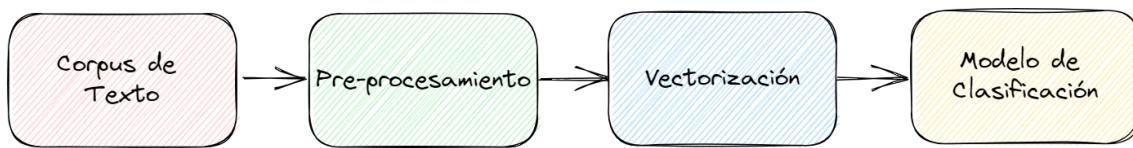
Las etiquetas de relaciones de dependencia según las especificaciones del proyecto [Universal Dependencies](#). Aquí vemos las Etiquetas de relaciones de dependencia (DepRel):

- **nsubj**: (Nominal Subject) Sujeto nominal de un verbo. Por ejemplo, en "El autobot está pensado", "El autobot" es el sujeto nominal de "está pensado".
- **obj**: (Object) Objeto de un verbo. Por ejemplo, en "evitar tareas tediosas", "tareas tediosas" es el objeto de "evitar".
- **amod**: (Adjectival Modifier) Un adjetivo que modifica un sustantivo. Por ejemplo, en "aspecto humanoide", "humanoide" es el modificador adjetival de "aspecto".
- **nmod**: (Nominal Modifier) Un modificador nominal de un sustantivo. Por ejemplo, en "sede en Austin", "en Austin" es un modificador nominal de "sede".
- **advcl**: (Adverbial Clause Modifier) Un modificador que es una cláusula adverbial. Por ejemplo, en "pensado para evitar", "para evitar" es un modificador adverbial de cláusula de "pensado".
- **advm**: (Adverbial Modifier) Un adverbio que modifica una palabra. Por ejemplo, en "ya realiza", "ya" es un modificador adverbial de "realiza".
- **cc**: (Coordinating Conjunction) Una conjunción coordinante. Por ejemplo, en "tediosas y agotadoras", "y" es una conjunción coordinante.
- **conj**: (Conjunct) Un conjunto que está vinculado a otro mediante una conjunción coordinante. Por ejemplo, en "tediosas y agotadoras", "agotadoras" es un conjunto con "tediosas".
- **appos**: (Appositional Modifier) Un modificador que está en aposición a otro. Por ejemplo, en "empresa Apptronik", "Apptronik" está en aposición a "empresa".
- **obl**: (Oblique) Un argumento no sujeto ni objeto de un verbo. Por ejemplo, en "realiza tareas en una empresa", "en una empresa" es un oblicuo de "realiza".
- **flat**: (Flat Multiword Expression) Una expresión de varias palabras que se agrupan en una sola unidad. Por ejemplo, en "Austin, Texas", "Texas" forma una expresión multi-palabra plana con "Austin".
- **mark**: (Marker) Un marcador de una cláusula subordinada. Por ejemplo, en "para evitar", "para" es un marcador.
- **case**: (Case Marking) Una palabra que marca el caso gramatical de otra palabra. Por ejemplo, en "de la empresa", "de" es una marca de caso para "empresa".
- **root**: (Root) La raíz del árbol de dependencia, generalmente el verbo principal de la oración.

- **cop:** (Copula) Una copula que funciona para vincular el sujeto con el predicado. Por ejemplo, en "está pensado", "está" es una copula.
- **det:** (Determiner) Un determinante que modifica un sustantivo. Por ejemplo, en "la empresa", "la" es el determinante de "empresa".
- **punct:** (Punctuation) Un token de puntuación.

## 4. Clasificación de texto (Text classification)

En este sección, veremos como clasificar texto, de modo que podamos asignar una categoría a una frase o un documento. Nuestro “pipeline” para el entrenamiento de un modelo clasificador, será el siguiente:



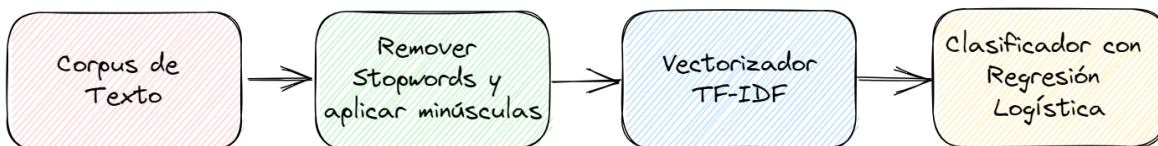
La clasificación de texto es una tarea de procesamiento del lenguaje natural (NLP) que involucra la asignación de una o más categorías o etiquetas predefinidas a fragmentos de texto. Estas etiquetas pueden representar diferentes temas, sentimientos, intenciones, etc. La clasificación de texto se utiliza en una amplia variedad de aplicaciones, incluyendo el filtrado de spam, análisis de sentimientos, etiquetado automático de contenido, y más.

El proceso general para construir un clasificador de texto incluye los siguientes pasos:

1. **Recolección de Datos:** Obtención de un conjunto de datos etiquetado que contenga ejemplos de cada clase que deseamos identificar.
2. **Preprocesamiento de Datos:** Limpiar y preparar los datos para el entrenamiento. Esto puede incluir la eliminación de ruido, normalización de texto, y otras técnicas para mejorar la calidad de los datos.
3. **Vectorización:** Transformar el texto en una representación numérica que pueda ser entendida y utilizada por un modelo de aprendizaje automático.
4. **Entrenamiento del Modelo:** Utilizar un algoritmo de aprendizaje automático para entrenar un modelo usando los datos preprocesados y vectorizados.
5. **Evaluación del Modelo:** Evaluar el rendimiento del modelo utilizando métricas adecuadas (como precisión, recall, F1-score, etc.) y un conjunto de datos de prueba separado.
6. **Implementación:** Desplegar el modelo entrenado en una aplicación real para clasificar textos nuevos y no etiquetados.

### Ejemplo con TF-IDF y Regresión Logística

En el ejemplo que veremos a continuación, utilizaremos `TfidfVectorizer` y `LogisticRegression` de Scikit-Learn:



- **TF-IDF como Vectorizador:** Usaremos el vectorizador TF-IDF (Frecuencia de Término - Frecuencia Inversa de Documento) para transformar los textos en una representación numérica. TF-IDF es una técnica que refleja la importancia de una palabra en un documento en relación con un conjunto de documentos (corpus). El vectorizador también elimina las palabras vacías ("stop words") en español para reducir el ruido y centrarse en las palabras más significativas.
- **Regresión Logística como Clasificador:** Hemos elegido la regresión logística como nuestro algoritmo de clasificación. Este algoritmo modela la relación entre características (las palabras en nuestro caso) y una variable categórica dependiente (las etiquetas de clase) utilizando una función logística.

```

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
import nltk

# Descargamos los stopwords que necesitaremos luego
nltk.download('stopwords')
from nltk.corpus import stopwords

# Obtenemos las stopwords para español
spanish_stop_words = stopwords.words('spanish')

labels = [(0, "desarrollo de software"), (1, "videojuegos"), (2, "inteligencia artificial"),
          (3, "ciberseguridad")]

dataset = []
# textos de "desarrollo de software"
dataset.append((0, "Me encanta programar en Python."))
dataset.append((0, "Python es un lenguaje versátil."))
dataset.append((0, "La programación en Java también es popular."))
dataset.append((0, "Ruby es otro lenguaje de programación interesante."))
dataset.append((0, "El desarrollo web es muy demandado actualmente."))
dataset.append((0, "JavaScript es esencial para el desarrollo web."))
dataset.append((0, "HTML y CSS son la base del desarrollo web."))
dataset.append((0, "El desarrollo frontend se complementa con el backend."))
dataset.append((0, "Los frameworks facilitan el desarrollo de software."))
dataset.append((0, "Git es una herramienta esencial para el control de versiones en el desarrollo de software."))
dataset.append((0, "La documentación es una parte crucial del desarrollo de software."))
dataset.append((0, "El testing es necesario para asegurar la calidad del software."))
dataset.append((0, "Los lenguajes más usados son Python, Java, JavaScript, C# y PHP."))
dataset.append((0, "Estos son ejemplos de lenguajes de bajo nivel: C, C++ y ensamblador."))

# textos de "videojuegos"
dataset.append((1, "Disfruto jugando videojuegos."))
dataset.append((1, "La realidad virtual es el futuro de los videojuegos."))
dataset.append((1, "Los videojuegos para móviles está en auge."))
dataset.append((1, "Los videojuegos indie han ganado mucha popularidad."))
dataset.append((1, "Las consolas de videojuegos son muy populares."))
dataset.append((1, "Los videojuegos 3D requieren una buena tarjeta gráfica."))
dataset.append((1, "Los videojuegos de estrategia son muy divertidos."))
dataset.append((1, "Una GPU potente es esencial para jugar algunos videojuegos."))
dataset.append((1, "Un buen joystick es esencial para jugar videojuegos."))
dataset.append((1, "Las aventuras gráficas son un género de videojuegos."))
dataset.append((1, "Un buen simulador de vuelo requiere un buen joystick."))

# textos de "inteligencia artificial"
dataset.append((2, "La inteligencia artificial transformará muchas industrias."))
dataset.append((2, "La robótica es una aplicación de la inteligencia artificial."))
dataset.append((2, "Las redes neuronales son un concepto clave en IA."))
dataset.append((2, "El aprendizaje profundo es una rama del aprendizaje automático."))
dataset.append((2, "El aprendizaje supervisado es un tipo de aprendizaje automático."))
dataset.append((2, "Las redes neuronales profundas son utilizadas en el aprendizaje profundo."))
dataset.append((2, "El aprendizaje por refuerzo es una técnica de aprendizaje automático."))
dataset.append((2, "El aprendizaje automático es fascinante."))
dataset.append((2, "El aprendizaje automático es una rama de la inteligencia artificial."))
dataset.append((2, "Los perceptrones son un concepto clave en las redes neuronales."))
dataset.append((2, "Una red convolucional es un tipo de red neuronal."))
dataset.append((2, "Las redes neuronales recurrentes son un tipo de red neuronal."))
dataset.append((2, "Las redes profundas son buenas para el reconocimiento de imágenes."))

# textos de "ciberseguridad"
dataset.append((3, "La ciberseguridad es crucial en el mundo digital."))
dataset.append((3, "La protección de datos personales es una parte importante de la ciberseguridad."))
dataset.append((3, "Los firewalls ayudan a proteger las redes corporativas."))
dataset.append((3, "La criptografía es una herramienta esencial en ciberseguridad."))
dataset.append((3, "La autenticación de dos factores es una técnica de ciberseguridad."))
dataset.append((3, "La ingeniería social es una técnica de hacking."))
dataset.append((3, "El phishing es una técnica de hacking."))
dataset.append((3, "El malware es un tipo de software malicioso."))
dataset.append((3, "El ransomware es un tipo de malware."))
dataset.append((3, "El spyware es un tipo de malware."))
dataset.append((3, "El adware es un tipo de malware."))
dataset.append((3, "El phishing es un tipo de ataque de ingeniería social."))
dataset.append((3, "El hacking ético es una profesión muy demandada."))
dataset.append((3, "Los hackers éticos ayudan a proteger los sistemas informáticos."))

# Preparar X e y
X = [text.lower() for label, text in dataset]
y = [label for label, text in dataset]

# División del dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

```

```

# Vectorización de los textos con eliminación de palabras vacías
vectorizer = TfidfVectorizer(stop_words=spanish_stop_words)
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)

# Creación y entrenamiento del modelo de Regresión Logística con multinomial
modelo_LR = LogisticRegression(max_iter=1000, multi_class='multinomial', solver='lbfgs')
modelo_LR.fit(X_train_vectorized, y_train)

# Evaluación del modelo de Regresión Logística
y_pred_LR = modelo_LR.predict(X_test_vectorized)
acc_LR = accuracy_score(y_test, y_pred_LR)
report_LR = classification_report(y_test, y_pred_LR, zero_division=1)

print("Precisión Regresión Logística:", acc_LR)
print("Reporte de clasificación Regresión Logística:\n", report_LR)

```

Y obtendremos como resultado, las métricas sobre la precisión de nuestro modelo:

```

Precisión Regresión Logística: 0.9090909090909091
Reporte de clasificación Regresión Logística:
      precision    recall   f1-score   support
          0         1.00     0.75     0.86       4
          1         1.00     1.00     1.00       2
          2         0.50     1.00     0.67       1
          3         1.00     1.00     1.00       4

      accuracy                           0.91      11
   macro avg       0.88     0.94     0.88      11
weighted avg       0.95     0.91     0.92      11

```

Una vez que tenemos nuestro modelo entrenado, podemos realizar inferencia, de modo que podamos clasificar nuevo texto:

```

# Definimos una lista de frases para clasificar
nuevas_frases = [
    "Los domingos suelo jugar videojuegos.",
    "La inteligencia artificial es fascinante.",
    "Los delitos informáticos son un flagelo cada vez más preocupante.",
    "Me gusta programar en Rust.",
    "La robótica suele utilizar inteligencia artificial.",
]

# Convertimos las frases a minúsculas
nuevas_frases = [frase.lower() for frase in nuevas_frases]

# Transformamos las nuevas frases usando el vectorizador que usamos para entrenar el modelo
nuevas_frases_vectorizadas = vectorizer.transform(nuevas_frases)

# Usamos el modelo entrenado para predecir las etiquetas de las nuevas frases
etiquetas_predichas = modelo_LR.predict(nuevas_frases_vectorizadas)

# Imprimimos las etiquetas predichas
for i, etiqueta in enumerate(etiquetas_predichas):
    print(f"La frase '{nuevas_frases[i]}' pertenece a la categoría: {labels[etiqueta][1]}")

```

Si quisieramos entender cuáles son las palabras que más influyen en nuestro clasificador para definir una categoría determinada, podríamos utilizar los coeficientes obtenidos con nuestro modelo:

```

import numpy as np
import matplotlib.pyplot as plt

# Obtén los nombres de las características y los coeficientes
feature_names = vectorizer.get_feature_names_out()
coef = modelo_LR.coef_

# Visualiza los coeficientes más importantes para cada clase
num_top_features = 10
for i, label in labels:
    top_features_idx = np.argsort(coef[i])[-num_top_features:]
    top_features_names = [feature_names[j] for j in top_features_idx]
    top_features_coef = coef[i][top_features_idx]

    plt.figure()
    plt.barh(top_features_names, top_features_coef)

```

```
plt.title(f'Características principales para la clase {label}')
plt.show()
```

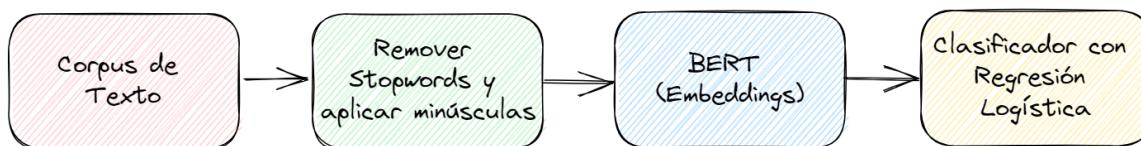
Y obtendríamos algo como esto para la categoría de “desarrollo de software”:



Esta visualización nos permite entender mejor como funciona el clasificador, e incluso realizar ajustes a nuestro dataset de entrenamiento si es necesario, para mejorar el rendimiento.

### Ejemplo con BERT como modelo de vectorización

Anteriormente usamos TF-IDF como método de vectorización. Pero también podemos utilizar modelos de embeddings para convertir nuestro texto en vectores.



Adaptaremos nuestro ejemplo anterior, para incorporar BERT en reemplazo de TF-IDF. Nuestro nuevo código, aplicando BERT será el siguiente:

```
# !pip install transformers

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import nltk
from transformers import BertTokenizer, BertModel
import torch
import numpy as np

# Cargar el tokenizador y modelo preentrenado de BERT para español
model_name = 'dccuchile/bert-base-spanish-wwm-cased'
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertModel.from_pretrained(model_name)

def get_bert_embeddings(texts):
    """Función para obtener los embeddings de BERT para una lista de textos."""
    embeddings = []
    for text in texts:
        inputs = tokenizer(text, return_tensors='pt', truncation=True, max_length=512)
        with torch.no_grad():
            outputs = model(**inputs)
            # Usamos el embedding del token [CLS] como la representación del texto
            embeddings.append(outputs.last_hidden_state[0][0].numpy())
    return np.array(embeddings)

# Descargamos los stopwords que necesitaremos luego
nltk.download('stopwords')
from nltk.corpus import stopwords

# Obtenemos las stopwords para español
spanish_stop_words = stopwords.words('spanish')
```

```

labels = [(0, "desarrollo de software"), (1, "videojuegos"), (2, "inteligencia artificial"),
          (3, "ciberseguridad")]

dataset = []
# textos de "desarrollo de software"
dataset.append((0, "Me encanta programar en Python."))
dataset.append((0, "Python es un lenguaje versátil."))
dataset.append((0, "La programación en Java también es popular."))
dataset.append((0, "Ruby es otro lenguaje de programación interesante."))
dataset.append((0, "El desarrollo web es muy demandado actualmente."))
dataset.append((0, "JavaScript es esencial para el desarrollo web."))
dataset.append((0, "HTML y CSS son la base del desarrollo web."))
dataset.append((0, "El desarrollo frontend se complementa con el backend."))
dataset.append((0, "Los frameworks facilitan el desarrollo de software."))
dataset.append((0, "Git es una herramienta esencial para el control de versiones en el desarrollo de software."))
dataset.append((0, "La documentación es una parte crucial del desarrollo de software."))
dataset.append((0, "El testing es necesario para asegurar la calidad del software."))

# textos de "videojuegos"
dataset.append((1, "Disfruto jugando videojuegos."))
dataset.append((1, "La realidad virtual es el futuro de los videojuegos."))
dataset.append((1, "Los videojuegos para móviles está en auge."))
dataset.append((1, "Los videojuegos indie han ganado mucha popularidad."))
dataset.append((1, "Las consolas de videojuegos son muy populares."))
dataset.append((1, "Los videojuegos 3D requieren una buena tarjeta gráfica."))
dataset.append((1, "Los videojuegos de estrategia son muy divertidos."))
dataset.append((1, "Una GPU potente es esencial para jugar algunos videojuegos."))
dataset.append((1, "Un buen joystick es esencial para jugar videojuegos."))
dataset.append((1, "Las aventuras gráficas son un género de videojuegos."))
dataset.append((1, "Un buen simulador de vuelo requiere un buen joystick."))

# textos de "inteligencia artificial"
dataset.append((2, "La inteligencia artificial transformará muchas industrias."))
dataset.append((2, "La robótica es una aplicación de la inteligencia artificial."))
dataset.append((2, "Las redes neuronales son un concepto clave en IA."))
dataset.append((2, "El aprendizaje profundo es una rama del aprendizaje automático."))
dataset.append((2, "El aprendizaje supervisado es un tipo de aprendizaje automático."))
dataset.append((2, "Las redes neuronales profundas son utilizadas en el aprendizaje profundo."))
dataset.append((2, "El aprendizaje por refuerzo es una técnica de aprendizaje automático."))
dataset.append((2, "El aprendizaje automático es fascinante."))
dataset.append((2, "El aprendizaje automático es una rama de la inteligencia artificial."))
dataset.append((2, "Los perceptrones son un concepto clave en las redes neuronales."))
dataset.append((2, "Una red convolucional es un tipo de red neuronal."))
dataset.append((2, "Las redes neuronales recurrentes son un tipo de red neuronal."))
dataset.append((2, "Las redes profundas son buenas para el reconocimiento de imágenes."))

# textos de "ciberseguridad"
dataset.append((3, "La ciberseguridad es crucial en el mundo digital."))
dataset.append((3, "La protección de datos personales es una parte importante de la ciberseguridad."))
dataset.append((3, "Los firewalls ayudan a proteger las redes corporativas."))
dataset.append((3, "La criptografía es una herramienta esencial en ciberseguridad."))
dataset.append((3, "La autenticación de dos factores es una técnica de ciberseguridad."))
dataset.append((3, "La ingeniería social es una técnica de hacking."))
dataset.append((3, "El phishing es una técnica de hacking."))
dataset.append((3, "El malware es un tipo de software malicioso."))
dataset.append((3, "El ransomware es un tipo de malware."))
dataset.append((3, "El spyware es un tipo de malware."))
dataset.append((3, "El adware es un tipo de malware."))
dataset.append((3, "El phishing es un tipo de ataque de ingeniería social."))
dataset.append((3, "El hacking ético es una profesión muy demandada."))
dataset.append((3, "Los hackers éticos ayudan a proteger los sistemas informáticos."))

# Preparar X e y
X = [text.lower() for label, text in dataset]
y = [label for label, text in dataset]

# División del dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Obtenemos los embeddings de BERT para los conjuntos de entrenamiento y prueba
X_train_vectorized = get_bert_embeddings(X_train)
X_test_vectorized = get_bert_embeddings(X_test)

# Creación y entrenamiento del modelo de Regresión Logística Multinomial
modelo_LR = LogisticRegression(max_iter=1000, multi_class='multinomial', solver='lbfgs')
modelo_LR.fit(X_train_vectorized, y_train)

# Evaluación del modelo de Regresión Logística
y_pred_LR = modelo_LR.predict(X_test_vectorized)
acc_LR = accuracy_score(y_test, y_pred_LR)
report_LR = classification_report(y_test, y_pred_LR, zero_division=1)

print("Precisión Regresión Logística:", acc_LR)

```

```

print("Reporte de clasificación Regresión Logística:\n", report_LR)

# Nuevas frases para clasificar
new_phrases = [
    "Quisiera aprender a programar en Rust.",
    "Los videojuegos de realidad virtual son increíbles.",
    "La inteligencia artificial está avanzando rápidamente."
]

# Preprocesamiento y vectorización de las nuevas frases
new_phrases_lower = [text.lower() for text in new_phrases]
new_phrases_vectorized = get_bert_embeddings(new_phrases_lower)

# Haciendo predicciones con el modelo entrenado
new_predictions = modelo_LR.predict(new_phrases_vectorized)

# Mostrando las predicciones junto con las frases
for text, label in zip(new_phrases, new_predictions):
    print(f"Texto: '{text}'")
    print(f"Clasificación predicha: {labels[label][1]}\n")

```

## 5. Análisis de sentimientos (Sentiment analysis)

El análisis de sentimiento, también conocido como minería de opiniones, es un subcampo del procesamiento del lenguaje natural (NLP, por sus siglas en inglés) que se centra en analizar, entender y obtener información sobre los sentimientos, opiniones o emociones expresadas en un texto. Esencialmente, el objetivo es determinar la actitud, el tono u otras características emocionales del texto.

### Aspectos clave del análisis de sentimiento:

#### 1. Polaridad:

- **Positivo:** El texto expresa una opinión favorable o un sentimiento positivo hacia el tema en cuestión.
- **Negativo:** El texto expresa una opinión desfavorable o un sentimiento negativo hacia el tema en cuestión.
- **Neutral:** El texto no expresa una opinión clara o tiene un sentimiento neutral hacia el tema.

#### 2. Intensidad:

- La fuerza del sentimiento expresado, que a menudo se cuantifica en una escala (por ejemplo, de 1 a 5).

#### 3. Aspecto:

- **Basado en aspectos:** El análisis de sentimientos que se centra en los diferentes aspectos o características de un producto o servicio.

### Técnicas comunes para el análisis de sentimiento:

#### 1. Basado en lexicones:

- Utilizar un diccionario predefinido de palabras, cada una con una puntuación de sentimiento asignada.

#### 2. Machine learning:

- Utilizar técnicas de aprendizaje automático (como regresión logística, máquinas de vectores de soporte, redes neuronales, etc.) para aprender patrones de sentimiento a partir de datos etiquetados.

#### 3. Deep learning:

- Utilizar modelos de aprendizaje profundo o transformers (como BERT, GPT, etc.) que son capaces de capturar relaciones semánticas complejas y entender el contexto para un análisis más preciso.

### Aplicaciones:

- **Análisis de productos:** Para entender la percepción del consumidor hacia productos o servicios.
- **Análisis de redes sociales:** Para monitorizar el sentimiento hacia una marca o tema en las redes sociales.
- **Servicio al cliente:** Para analizar los comentarios de los clientes y mejorar el servicio.
- **Análisis de mercado:** Para realizar análisis de mercado y de la competencia.

### Desafíos:

- **Sarcasmo e ironía:** Dificultad para detectar el sarcasmo y la ironía, ya que requiere un entendimiento profundo del lenguaje.
- **Ambigüedad:** Los textos pueden ser ambiguos y pueden tener más de un significado, lo que complica el análisis.

Veamos un ejemplo utilizando una librería `sentiment-spanish`. El modelo se basa en Machine Learning y utiliza el método `CountVectorizer` de scikit-learn para la vectorización de las características de los textos. Este vectorizador convierte la colección de documentos de texto en una matriz de conteos de tokens. Posteriormente, el modelo utiliza un clasificador `MultinomialNB`, que es un clasificador Naive Bayes multinomial de scikit-learn, para llevar a cabo la clasificación de los textos según su sentimiento. Este clasificador se entrena utilizando los vectores de características obtenidos mediante `CountVectorizer`.

```
# !pip install sentiment-analysis-spanish
from sentiment_analysis_spanish import sentiment_analysis
sentiment = sentiment_analysis.SentimentAnalysisSpanish()
print(sentiment.sentiment("me gusta la fiesta, es fabulosa"))
# Imprime: 0.8322652664199587
```

El modelo predice una puntuación de sentimiento que va de 0 a 1, donde valores cercanos a 0 representan un sentimiento negativo y valores cercanos a 1 representan un sentimiento positivo. Fue entrenado utilizando más de 800,000 reseñas de usuarios de las páginas eltenedor, decathlon, tripadvisor, filmaffinity y ebay, recopiladas a través de web scraping.

Veamos otro ejemplo, utilizando un modelo basado en BERT, el cual es multilingüe (Inglés, Holandés, Alemán, Francés, Español):

```
# !pip install transformers
from transformers import BertTokenizer, BertForSequenceClassification
from transformers import pipeline

# Cargamos el tokenizador y el modelo
model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name)

# Creamos un pipeline de clasificación
nlp = pipeline("sentiment-analysis", model=model, tokenizer=tokenizer)

# Lista de frases para analizar
frases = [
    "Me gusta mucho este producto.",
    "El servicio fue terrible, no estoy nada contento.",
    "The food was delicious, I will definitely come back again.",
    "El lugar está un poco descuidado y sucio."
]

# Obtenemos las predicciones de sentimiento para cada frase
for frase in frases:
    result = nlp(frase)
    print(f"Frase: '{frase}'")
    print(f"Sentimiento: {result[0]['label']}, Score: {result[0]['score']:.3f}")
    print()
```

Y el resultado será:

```
Frage: 'Me gusta mucho este producto.'
Sentimiento: 5 stars, Score: 0.493

Frage: 'El servicio fue terrible, no estoy nada contento.'
Sentimiento: 1 star, Score: 0.825

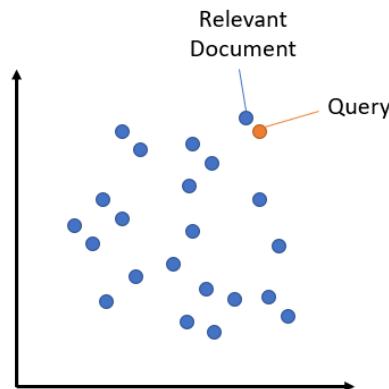
Frage: 'The food was delicious, I will definitely come back again.'
Sentimiento: 5 stars, Score: 0.525

Frage: 'El lugar está un poco descuidado y sucio.'
Sentimiento: 3 stars, Score: 0.627
```

## 6. Búsqueda semántica

La idea detrás de la búsqueda semántica es realizar embeddings de los textos de un corpus, ya sean oraciones, párrafos o documentos, en un espacio vectorial.

A la hora de realizar la búsqueda, se realiza el embedding de la pregunta o consulta en el mismo espacio vectorial de los textos, y luego buscamos los vectores más cercanos entre la consulta y los documentos del corpus:



### Búsqueda Semántica Simétrica

En la búsqueda semántica simétrica, tanto la consulta como las entradas en el corpus tienen aproximadamente la misma longitud y cantidad de contenido. Esto significa que están en un terreno bastante equitativo en términos de la cantidad de información que contienen.

#### Características

- **Longitud similar de textos:** Los textos de la consulta y del corpus tienen longitudes similares, lo que facilita la comparación directa.
- **Intercambiabilidad:** Potencialmente, se podría invertir la consulta y las entradas del corpus, manteniendo una coherencia en los resultados de la búsqueda.

#### Ejemplo

Si tu consulta es "¿Cómo aprender Python en línea?", querrías encontrar una entrada como "Aprendiendo Python en la web".

### Búsqueda Semántica Asimétrica

En la búsqueda semántica asimétrica, generalmente se tiene una consulta corta, como una pregunta o algunas palabras clave, y se busca encontrar un párrafo más largo que responda a la consulta.

#### Características

- **Diferencia significativa en la longitud de los textos:** La consulta es generalmente más corta que las entradas del corpus que está buscando.
- **No intercambiabilidad:** Invertir la consulta y las entradas del corpus generalmente no tiene sentido debido a la disparidad en la longitud y el contenido.

#### Ejemplo

Si tu consulta es "¿Qué es Python?", querrías encontrar un párrafo que explique qué es Python, como "Python es un lenguaje de programación interpretado, de alto nivel y de propósito general..."

Hay modelos especialmente pre-entrenados para búsqueda asimétrica (por ejemplo [Pre-Trained MS MARCO Models](#)). [MS MARCO](#) es un corpus de recuperación de información a gran escala que fue creado basándose en consultas de búsqueda reales de usuarios utilizando el motor de búsqueda Microsoft Bing. Los modelos proporcionados pueden ser utilizados para búsqueda semántica, es decir, dado unas palabras clave / una frase de búsqueda / una pregunta, el modelo encontrará pasajes que son relevantes para la consulta de búsqueda.

Los datos de entrenamiento constan de más de 500 mil ejemplos, mientras que el corpus completo consta de más de 8.8 millones de pasajes.

Veamos un ejemplo, utilizando un [modelo específico de preguntas y respuestas basado en S-BERT](#):

```

# !pip install sentence-transformers

from sentence_transformers import SentenceTransformer, util
modelo = SentenceTransformer('msmarco-MiniLM-L-6-v3')

# Lista de consultas y respuestas
consultas = [
    '¿Qué tan grande es Londres?',
    '¿Cuál es la capital de Francia?',
    '¿Qué es la fotosíntesis?',
    '¿De qué depende la vida en el planeta?',
    '¿Cuál fue el rey de los dinosaurios?',
    '¿Cuándo y dónde vivieron los Tiranosaurios?',
    '¿Cómo se extinguieron los dinosaurios?'
]

respuestas = [
    'La vida en la Tierra depende fundamentalmente de la energía solar. Esta energía es atrapada mediante la fotosíntesis, responsable de que los vegetales crezcan y se multipliquen. La fotosíntesis es un proceso mediante el cual las plantas, algas y algunas bacterias convierten la luz solar, dióxido de carbono y agua en glucosa y oxígeno.',
    'Londres tenía 9,787,426 habitantes según el censo de 2011',
    'La capital de Francia es París.',
    'El Tiranosaurio Rex es considerado el rey de los dinosaurios y la cultura popular lo ha posicionado como el dinosaurio más temible.',
    'El Tiranosaurio Rex vivió hace 68 y 66 millones de años en las últimas etapas del período Cretácico distribuidos por Norteamérica.',
    'Los dinosaurios se extinguieron de la faz de la tierra, debido al impacto de un gran meteorito.',
    'El alimento preferido de los conejos son las zanahorias.',
    'La fotosíntesis es un proceso mediante el cual las plantas, algas y algunas bacterias convierten la luz solar, dióxido de carbono y agua en glucosa y oxígeno.'
]

# Generar incrustaciones para todas las consultas y respuestas
incrustaciones_consultas = modelo.encode(consultas)
incrustaciones_respuestas = modelo.encode(respuestas)

# Encontrar la respuesta con la mejor similitud para cada consulta
for i, incrustacion_consulta in enumerate(incrustaciones_consultas):
    similitudes = util.cos_sim/incrustacion_consulta, incrustaciones_respuestas)[0]
    mejor_indice = similitudes.argmax()
    print(f"Consulta: {consultas[i]}")
    print(f"Mejor respuesta (Similitud: {similitudes[mejor_indice]:.4f}): {respuestas[mejor_indice]}")
    print()

```

La salida de nuestro ejemplo, será:

```

Consulta: ¿Qué tan grande es Londres?
Mejor respuesta (Similitud: 0.4851): Londres tenía 9,787,426 habitantes según el censo de 2011

Consulta: ¿Cuál es la capital de Francia?
Mejor respuesta (Similitud: 0.7379): La capital de Francia es París.

Consulta: ¿Qué es la fotosíntesis?
Mejor respuesta (Similitud: 0.6341): La fotosíntesis es un proceso mediante el cual las plantas, algas y algunas bacterias convierten la luz solar, dióxido de carbono y agua en glucosa y oxígeno.

Consulta: ¿De qué depende la vida en el planeta?
Mejor respuesta (Similitud: 0.6135): La vida en la Tierra depende fundamentalmente de la energía solar. Esta energía es atrapada mediante la fotosíntesis, responsable de que los vegetales crezcan y se multipliquen.

Consulta: ¿Cuál fue el rey de los dinosaurios?
Mejor respuesta (Similitud: 0.5627): El Tiranosaurio Rex es considerado el rey de los dinosaurios y la cultura popular lo ha posicionado como el dinosaurio más temible.

Consulta: ¿Cuándo y dónde vivieron los Tiranosaurios?
Mejor respuesta (Similitud: 0.5550): Los dinosaurios se extinguieron de la faz de la tierra, debido al impacto de un gran meteorito.

Consulta: ¿Cómo se extinguieron los dinosaurios?
Mejor respuesta (Similitud: 0.7098): Los dinosaurios se extinguieron de la faz de la tierra, debido al impacto de un gran meteorito.

```

También podemos utilizar `util.semantic_search` para obtener una lista de las mejores respuestas. Con el parámetro `top_k=3` obtendríamos las mejores 3 respuesta. Ejemplo:

```

# Encontrar las tres respuestas con mejor similitud para cada consulta usando util.semantic_search
for i, incrustacion_consulta in enumerate(incrustaciones_consultas):
    hits = util.semantic_search(incrustacion_consulta, incrustaciones_respuestas, top_k=3)
    print(f"Consulta: {consultas[i]}")
    for j, hit in enumerate(hits[0]):
        print(f"Respuesta {j+1} (Similitud: {hit['score']:.4f}): {respuestas[hit['corpus_id']]}")
    print()

```

Y obtendríamos por ejemplo:

Consulta: ¿Cómo se extinguieron los dinosaurios?  
Respuesta 1 (Similitud: 0.7098): Los dinosaurios se extinguieron de la faz de la tierra, debido al impacto de un gran meteorito.  
Respuesta 2 (Similitud: 0.4600): El Tiranosaurio Rex es considerado el rey de los dinosaurios y la cultura popular lo ha posicionado como el animal más grande que jamás ha existido.  
Respuesta 3 (Similitud: 0.4039): El alimento preferido de los conejos son las zanahorias.



Es este tipo de búsquedas, es habitual poner un umbral mínimo, para considerar solamente las respuestas que sean mayores a dicho umbral. De esa manera podemos evitar respuestas de baja probabilidad de ser correctas.

## Optimización de búsqueda

Para manejar grandes conjuntos de vectores y realizar búsquedas de similitud eficientes, se utilizan varias técnicas y herramientas especializadas. Existen herramientas como **Annoy** (<https://github.com/spotify/annoy>) y **FAISS** (<https://github.com/facebookresearch/faiss>) entre otras, que resultan muy eficientes en la búsqueda sobre grandes volúmenes de datos sobre bases de datos vectoriales. En la Unidad 5, veremos este tipo de técnicas.

## 7. Desambiguación (Disambiguation)

La desambiguación del sentido de las palabras (Word-Sense Disambiguation) es el proceso de identificar qué sentido de una palabra se pretende en una oración u otro segmento de contexto. En el procesamiento y cognición del lenguaje humano, generalmente es subconsciente/automático, pero a menudo puede llegar a ser consciente cuando la ambigüedad perjudica la claridad de la comunicación, dado el polisemia generalizada en el lenguaje natural. En la lingüística computacional, es un problema abierto que afecta al cómo los modelos interpretan correctamente el lenguaje y la intención de los usuarios.

La desambiguación es un proceso crucial en NLP que busca determinar el sentido correcto de una palabra en un contexto específico. Este proceso es vital para una variedad de aplicaciones, incluyendo traducción automática, recuperación de información, sistemas de respuesta a preguntas, análisis de sentimientos y comprensión del lenguaje natural.

### Significado de la desambiguación

La ambigüedad en el lenguaje puede llevar a malentendidos y errores en las aplicaciones de NLP. Por ejemplo, la palabra "gato" puede referirse al animal, como a un dispositivo para elevar un automóvil. La WSD ayuda a seleccionar el sentido correcto de una palabra, mejorando así la precisión y exactitud de las tareas de procesamiento del lenguaje.

### Desafíos en la WSD

La WSD es una tarea compleja debido a varios desafíos, incluyendo la ambigüedad léxica, la dependencia del contexto, la escasez de datos etiquetados para el entrenamiento, y la granularidad del sentido de las palabras. Además, la WSD en múltiples idiomas añade una capa adicional de complejidad, requiriendo recursos y técnicas multilingües.

### Enfoques para la WSD

Existen varios enfoques para abordar los desafíos de la WSD, incluyendo:

- **Métodos basados en conocimientos:** Estos enfoques aprovechan recursos léxicos como diccionarios, tesauros y ontologías para determinar los sentidos de las palabras. Se basan en reglas hechas a mano o relaciones semánticas entre palabras para desambiguar los sentidos.
- **Aprendizaje supervisado:** Utilizando datos de entrenamiento etiquetados, los algoritmos de aprendizaje supervisado, como Naïve Bayes, Máquinas de Vectores de Soporte (SVM) o Redes Neuronales, pueden ser entrenados para predecir los sentidos de las palabras basándose en características contextuales.
- **Aprendizaje no supervisado y semi-supervisado:** Estos enfoques aprovechan grandes cantidades de datos no etiquetados y explotan patrones, estadísticas de co-ocurrencia y técnicas de agrupación para agrupar contextos similares e inferir sentidos.
- **Embeddings de sentido:** Al aprender representaciones distribuidas de los sentidos de las palabras, las incrustaciones de sentido capturan relaciones semánticas y similitudes, permitiendo una WSD efectiva a través de enfoques basados en similitud.
- **Enfoques híbridos:** La combinación de múltiples técnicas, como los métodos basados en conocimientos y el aprendizaje automático, o los métodos supervisados y no supervisados, puede llevar a una mejora del rendimiento en la desambiguación.

Veamos un ejemplo de como aplicarla la desambiguación en Python, usando un modelo de aprendizaje supervisado basado en la arquitectura T5 ([Text-To-Text Transfer Transformer](#)). Lamentablemente, el modelo que usaremos fue entrenado en idioma inglés:

```
# !pip install transformers
# !pip install sentencepiece

from transformers import AutoModelForSeq2SeqLM, AutoTokenizer, Text2TextGenerationPipeline

# Modelo: https://huggingface.co/jpwahle/t5-large-word-sense-disambiguation
pipe = Text2TextGenerationPipeline(
    model = AutoModelForSeq2SeqLM.from_pretrained("jpelhaw/t5-word-sense-disambiguation"),
    tokenizer = AutoTokenizer.from_pretrained("jpelhaw/t5-word-sense-disambiguation")
)

input = (
"""
question: which description describes the word " java " best in the following context? \
descriptions:[ " A drink consisting of an infusion of ground coffee beans " ,
                " a platform-independent programming language " \
                "or an island in Indonesia to the south of Borneo "
context: I like to drink " java " in the morning .
"""
)

output = pipe(input)[0]['generated_text']
print(output)

# Salida:
# a drink consisting of an infusion of ground coffee beans
```

Dentro de las opciones disponibles para desambiguar texto, tenemos también esta alternativa usando [nltk](#) junto con la librería [pywsd](#):

```
# Instalar pywsd
# !pip install pywsd

# Importar las funciones y módulos necesarios
import nltk
from nltk.corpus import wordnet as wn
from nltk.stem import PorterStemmer
from itertools import chain
from pywsd.lesk import simple_lesk

# Descargar los recursos necesarios de NLTK
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Definir las oraciones para la desambiguación
bank_sents = [
    'I went to the bank to deposit my money',
    'The river bank was full of dead fishes'
]

# Llamar a la función Lesk y imprimir los resultados para ambas oraciones
# Contexto y desambiguación para la primera oración
print("Context-1:", bank_sents[0])
answer = simple_lesk(bank_sents[0], 'bank')
print("Sense:", answer)
print("Definition:", answer.definition())

# Contexto y desambiguación para la segunda oración
print("Context-2:", bank_sents[1])
answer = simple_lesk(bank_sents[1], 'bank', 'n')
print("Sense:", answer)
print("Definition:", answer.definition())
```

Cuya salida será:

```
Context-1: I went to the bank to deposit my money
Sense: Synset('depository_financial_institution.n.01')
Definition: a financial institution that accepts deposits and channels the money into lending activities
```

```
Context-2: The river bank was full of dead fishes
Sense: Synset('bank.n.01')
Definition: sloping land (especially the slope beside a body of water)
```

Aquí hemos usado el [algoritmo de Lesk](#), que es un método de desambiguación del sentido de las palabras que se basa en el uso de definiciones de diccionario para establecer el sentido correcto de una palabra en un contexto específico. Fue desarrollado por [Michael Lesk](#) en la década de 1980. El algoritmo original de Lesk compara el conjunto de palabras en el contexto de la palabra ambigua con el conjunto de palabras en las definiciones de los posibles sentidos de esa palabra (generalmente obtenidas de un diccionario o una base de datos léxica como WordNet). El sentido que tiene la mayor cantidad de palabras en común con el contexto es el que se selecciona.

El algoritmo de Lesk se clasifica como un método basado en conocimientos porque utiliza recursos léxicos externos (como diccionarios y tesauros) para determinar el sentido de las palabras. No requiere datos de entrenamiento etiquetados, lo que lo distingue de los métodos de aprendizaje supervisado.

## 8. Resumen de texto (Text Summarization)

El "text summarization" o resumen automático de texto es una tarea dentro del Procesamiento del Lenguaje Natural (NLP, por sus siglas en inglés) que tiene como objetivo producir un resumen conciso y coherente de un texto más extenso. Esta tarea es esencial para procesar grandes cantidades de información y presentarla de manera comprensible y rápida al usuario. Hay dos enfoques principales para el resumen automático de texto:

### 1. Resumen Extractivo:

- Este enfoque selecciona y extrae frases o segmentos directamente del texto original para formar el resumen. Es decir, no se generan nuevas frases, sino que se eligen las más representativas del texto original.
- Se basa en la ponderación de la importancia de las frases, que puede determinarse mediante diversas técnicas, como la frecuencia de términos, la posición de la frase en el texto, entre otros.

### 2. Resumen Abstractivo:

- Genera un resumen reescribiendo partes del texto original con nuevas frases.
- Puede utilizar técnicas de modelado del lenguaje, como las redes neuronales recurrentes (RNN) o los modelos de atención como Transformer.
- Es más complejo que el resumen extractivo y puede generar resúmenes más naturales y coherentes, pero también es más propenso a errores o a generar información que no estaba en el texto original (alucinaciones).

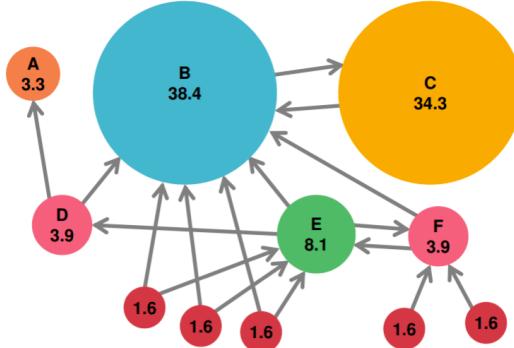
### Aplicaciones:

- Resumen de noticias o artículos.
- Generación de resúmenes para documentos académicos o informes.
- Herramientas de búsqueda y recomendación de contenido.

### Desafíos:

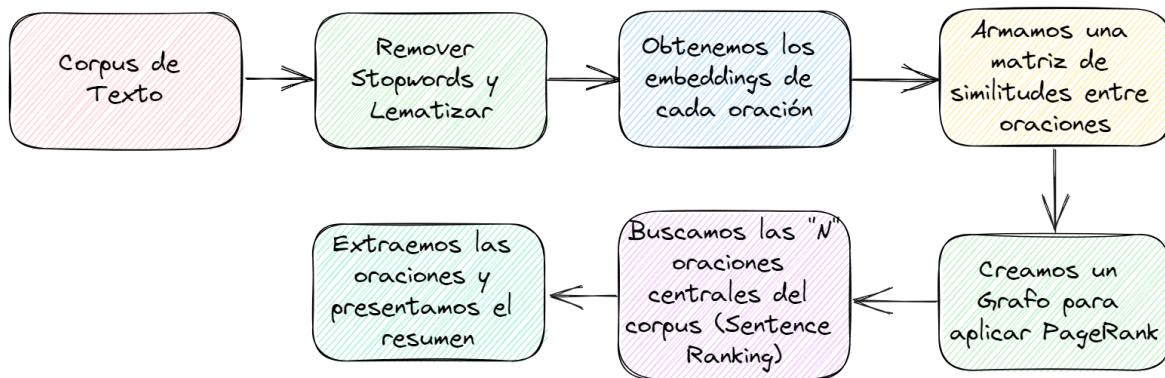
- Mantener la coherencia y relevancia en el resumen.
- Evitar la pérdida de información esencial.
- Generar resúmenes que sean fieles al contenido original sin introducir información errónea o sesgada.

Veamos a continuación un ejemplo de como realizar un resumen Extractivo. El procedimiento que haremos es conocido como TextRank, que es una adaptación de [PageRank](#) aplicado al NLP.



PageRank es una familia de algoritmos creada y desarrollada por la compañía tecnológica estadounidense Google para optimizar las búsquedas de páginas web, y es la base lógica sobre la que se fundamenta su motor de búsqueda, (Patente 1999)

Los pasos que haremos son los siguientes:



Y aquí lo implementamos usando `spacy` y `networkx`. Spacy nos permite resolver varias cosas:

- Cargar un modelo pre-entrenado similar a Word2Vec, que permite obtener embeddings de palabras y frases. En este último caso, realiza la media de los vectores de palabras que conforman la oración.
- Lematizar el texto
- Quitar las stopwords

Con `networkx` haremos el grafo y aplicaremos `pagerank`

```

# !pip install spacy
# !python -m spacy download es_core_news_md

import spacy
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import networkx as nx
from sklearn.metrics.pairwise import cosine_similarity

# Función para generar un resumen extractivo usando PageRank
def summarize(similarity_matrix, num_sentences=5):
    # Crear un grafo a partir de la matriz de similitud
    nx_graph = nx.from_numpy_array(similarity_matrix)
    # Aplicar PageRank al grafo
    scores = nx.pagerank(nx_graph)
    # Ordenar las oraciones por su puntuación y seleccionar las mejores
    ranked_sentences = sorted([(scores[i], s) for i, s in enumerate(original_sentences)], reverse=True)
    return ' '.join([ranked_sentences[i][1] for i in range(num_sentences)])

# Cargar el modelo de spaCy
nlp = spacy.load('es_core_news_md')

# Texto de ejemplo
text = """
Ciencia y Tecnología.\nEn esta sección, hablaremos de algo muy interesante. La inteligencia artificial (IA) se refiere a la simulación
"""

# Tokenizar el texto
tokens = nlp(text)

# Extraer las oraciones
sentences = [str(token) for token in tokens if token.is_sentence]

# Crear una lista de vectores embeddings para cada oración
embeddings = [nlp(sentence).vector for sentence in sentences]

# Calcular la matriz de similitud entre las oraciones
similarity_matrix = np.array([[cosine_similarity(embeddings[i].reshape(1, -1), embeddings[j].reshape(1, -1))[0] for j in range(len(embeddings))]] for i in range(len(embeddings)))

# Generar el resumen
summary = summarize(similarity_matrix, num_sentences=5)
print(summary)
  
```

```

doc = nlp(text)

# Lematizar y eliminar stopwords de cada oración
lemmatized_sentences = []
original_sentences = []
for sent in doc.sents:
    lemmatized_sentence = " ".join([token.lemma_ for token in sent if not token.is_stop and not token.is_punct])
    if lemmatized_sentence.strip() != '': # Asegurarse de que la oración no esté vacía
        lemmatized_sentences.append(lemmatized_sentence)
        original_sentences.append(str(sent).strip())

# Procesar las oraciones lematizadas con spaCy para obtener sus vectores
lemmatized_docs = [nlp(sent) for sent in lemmatized_sentences]

# Obtenemos una lista con los vectores de cada oración
sentence_vectors = [sent.vector for sent in lemmatized_docs]

# Crear una matriz de similitud entre las oraciones filtradas
similarity_matrix = cosine_similarity(sentence_vectors)

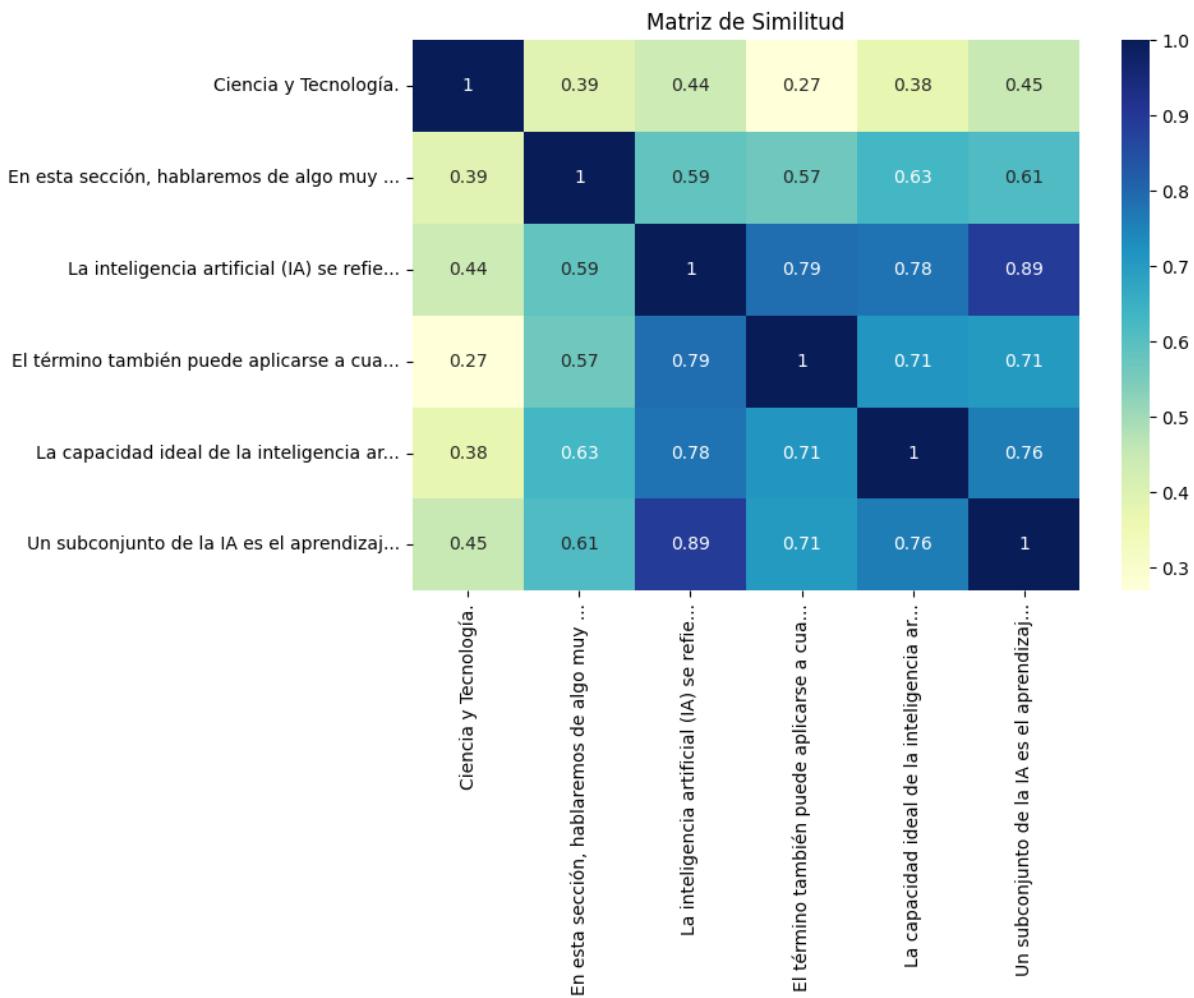
# Acortar las oraciones originales para usarlas como etiquetas en el gráfico
sentence_labels_short = []
for sent in original_sentences:
    if len(sent) > 40:
        sent = sent[:40] + '...'
    sentence_labels_short.append(sent)

# Graficar la matriz de similitud usando un mapa de calor
plt.figure(figsize=(10, 8))
sns.heatmap(similarity_matrix, annot=True, cmap='YlGnBu', xticklabels=sentence_labels_short, yticklabels=sentence_labels_short)
plt.title("Matriz de Similitud")
plt.xticks(rotation=90) # Rotar las etiquetas del eje x para mejorar la legibilidad
plt.yticks(rotation=0) # Puedes ajustar la rotación de las etiquetas del eje y si es necesario
plt.tight_layout() # Ajustar el layout para que todo encaje bien
plt.show()

# Generar resumen extractivo
resumen = summarize(similarity_matrix, num_sentences=2)
print("\nResumen Extractivo:")
print(resumen)

```

Y el resultado será:



#### Resumen Extractivo:

La inteligencia artificial (IA) se refiere a la simulación de la inteligencia humana en máquinas que están programadas para pensar y actuar de forma similar a los humanos.

Estas son las partes más importantes del código del ejemplo:

#### Creación del Grafo

`nx_graph = nx.from_numpy_array(similarity_matrix)` : Se crea un grafo a partir de la matriz de similitud utilizando `networkx`. Cada nodo del grafo representa una oración del texto, y los edges (conexiones) entre los nodos están ponderados por la similitud entre las oraciones.

#### Aplicación de PageRank:

`scores = nx.pagerank(nx_graph)` : Se aplica el algoritmo PageRank al grafo. Esto devuelve un diccionario donde las claves son los nodos (oraciones) y los valores son los scores de PageRank para cada nodo. Un score más alto indica que la oración es más "importante" o central en el texto.

#### Ordenación de Oraciones:

`ranked_sentences = sorted(...)` : Se ordenan las oraciones en función de sus scores de PageRank en orden descendente. Esto nos da una lista de oraciones desde la más importante hasta la menos importante.

Como resultado, las oraciones centrales seleccionadas, son las frases que tienen más cercanía con el resto de las frases del documento. En el ejemplo tomamos las 2 frases con mayor puntaje, y que mejor representan el contenido de todo el corpus.



Este método de resumen utilizado, se podría mejorar usando un modelo como Sentence-BERT para obtener los embeddings de la frase, en vez de usar el método de promediado de vectores que hace `spacy`, que como hemos mencionado anteriormente, pierde información de la frase.

Veamos ahora un ejemplo de como podemos realizar **resumen Abstractivo** en Python:

```
# Importamos las bibliotecas necesarias
import re
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

# Definimos una función para manejar y limpiar espacios en blanco y saltos de línea
WHITESPACE_HANDLER = lambda k: re.sub('\s+', ' ', re.sub('\n+', ' ', k.strip()))

# Texto del artículo que queremos resumir
article_text = """
El virus Nipah produjo un nuevo brote en la India. Se detectó en 6 personas y 2 de ellas murieron. Son del estado de Kerala, en el sur de la India. Aunque la infección por Nipah solo se ha diagnosticado en pocas personas, hoy es una amenaza preocupante. Está clasificado como patógeno de riesgo 2. Además, ha sido incluido por la Organización Mundial de la Salud (OMS) en el plan de investigación y desarrollo que identifica patógenos emergentes. Si bien los brotes de Nipah suelen afectar a una zona geográfica relativamente pequeña, pueden ser mortales, y a algunos científicos les preocupa que pueda propagarse más ampliamente.
"""

# Nombre del modelo que vamos a utilizar para el resumen
model_name = "csebuetnlp/t5_multilingual_XLSum"

# Cargamos el tokenizador y el modelo del nombre especificado
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

# Convertimos el texto del artículo en IDs de entrada usando el tokenizador
input_ids = tokenizer(
    [WHITESPACE_HANDLER(article_text)],
    return_tensors="pt",
    padding="max_length",
    truncation=True,
    max_length=512
)[["input_ids"]]

# Generamos el resumen usando el modelo
output_ids = model.generate(
    input_ids=input_ids,
    max_length=84,
    no_repeat_ngram_size=2,
    num_beams=4
)[0]

# Decodificamos los IDs de salida para obtener el resumen en texto
summary = tokenizer.decode(
    output_ids,
    skip_special_tokens=True,
    clean_up_tokenization_spaces=False
)

# Imprimimos el resumen
print(f'Artículo:\n{article_text}\n')
print(f'Resumen:\n{summary}\n')
```

Y el resultado será el siguiente:

```
Artículo:
El virus Nipah produjo un nuevo brote en la India. Se detectó en 6 personas y 2 de ellas murieron. Son del estado de Kerala, en el sur de la India. Aunque la infección por Nipah solo se ha diagnosticado en pocas personas, hoy es una amenaza preocupante. Está clasificado como patógeno de riesgo 2. Además, ha sido incluido por la Organización Mundial de la Salud (OMS) en el plan de investigación y desarrollo que identifica patógenos emergentes. Si bien los brotes de Nipah suelen afectar a una zona geográfica relativamente pequeña, pueden ser mortales, y a algunos científicos les preocupa que pueda propagarse más ampliamente.

Resumen:
Un nuevo brote de Nipah, el virus que afecta a la región de Kerala, en el suroeste de India, ha causado la muerte en la última semana.
```

Aquí hemos usado un modelo multilingüe (soporta 44 lenguajes) para realizar resumen de texto. El modelo se llama XL-Sum y ha sido entrenado con un dataset de la BBC, que consiste en extractos de noticias y sus correspondientes resúmenes.

Otra opción para resumen **Abstractivo**, es usar el modelo multipropósito T5. Este modelo considera todas las tareas de NLP como una conversión de texto a texto, lo que significa que tareas como traducción, resumen, y preguntas y respuestas se formulan de una manera en que la entrada y la salida son secuencias de texto. En este caso usamos t5-small, pero también existen otros tamaños de modelos (t5-base, t5-large, t5-3b, t5-11b) con diferentes cantidad de parámetros.

```
# !pip install transformers sentencepiece torch
from transformers import T5Tokenizer, T5ForConditionalGeneration
```

```

# Cargar el tokenizador y el modelo
tokenizer = T5Tokenizer.from_pretrained('t5-small')
model = T5ForConditionalGeneration.from_pretrained('t5-small')

# Definir el texto en español que queremos resumir
texto = """
La contaminación del aire es uno de los problemas ambientales más graves en el mundo moderno, afectando a millones de personas en todo el mundo. Las partículas contaminantes pueden causar una variedad de problemas de salud, incluyendo enfermedades respiratorias, cardiovasculares y cáncer. La exposición prolongada a la contaminación del aire puede reducir la esperanza de vida y afectar la calidad de vida de las personas. Es crucial tomar medidas para reducir la contaminación del aire y minimizar sus impactos en la salud humana y el medio ambiente.
"""

# Preprocesar el texto y generar un resumen
inputs = tokenizer.encode("summarize: " + texto, return_tensors="pt", max_length=512, truncation=True)
summary_ids = model.generate(inputs, max_length=100, min_length=25, length_penalty=2.0, num_beams=4, early_stopping=True)
resumen = tokenizer.decode(summary_ids[0], skip_special_tokens=True)

# Imprimir el resumen
print(resumen)

```

`max_length=100`: Esto establece la longitud máxima del resumen generado en 100 tokens.

`min_length=25`: Esto establece la longitud mínima del resumen generado en 25 tokens.

Al ejecutar obtenemos el siguiente resultado:

```
la contaminación del aire puede reducir la esperanza de vida y afectar la calidad de vida de las personas.
```

## 9. Detección de idioma (Language Detection)

En el procesamiento de lenguaje natural (NLP), la detección de idioma se refiere al proceso de determinar automáticamente en qué idioma está escrito un texto dado. Es un paso crucial en muchas aplicaciones de NLP, especialmente en sistemas multilingües y plataformas globales, donde los textos pueden estar disponibles en varios idiomas. Para poder realizar la detección, se utilizan distintos métodos, por ejemplo:

### 1. Frecuencia de Caracteres y Palabras:

- Algunos métodos de detección de idioma se basan en analizar la frecuencia de caracteres o palabras en el texto y compararla con las frecuencias características de varios idiomas.

### 2. Modelos Estadísticos:

- Otros métodos utilizan modelos estadísticos o de aprendizaje automático entrenados para reconocer la estructura lingüística y las peculiaridades de diferentes idiomas.

### 3. Modelos de Aprendizaje Profundo:

- En casos más avanzados, se pueden usar modelos de aprendizaje profundo para detectar el idioma de un texto, incluso en situaciones donde el texto contiene múltiples idiomas.

El algoritmo subyacente en `langdetect` construye perfiles de idiomas basados en n-gramas de caracteres y un modelo de tipo Naive Bayes (<https://www.slideshare.net/shuyo/language-detection-library-for-java>)

Veamos un ejemplo utilizando `langdetect` (<https://github.com/Mimino666/langdetect>), que resulta muy sencilla de utilizar:

```

# !pip install langdetect
from langdetect import detect

texto = "Escribe el texto del cual quieras detectar el idioma."
idioma = detect(texto)
print(idioma) # Imprime el código ISO 639-1 del idioma detectado, por ejemplo, 'es' para español.

texto = "J'aime lire des livres et écouter de la musique."
idioma = detect(texto)
print(idioma)

#Imprime:
# es
# fr

```

Algunas consideraciones:

- `langdetect` puede no ser siempre preciso, especialmente con textos cortos o textos que contienen múltiples idiomas.
- Puede haber variabilidad en los resultados; ejecutar la detección varias veces en el mismo texto puede dar diferentes resultados. Para reducir esta variabilidad, se puede usar el método `detect_langs()`, que devuelve una lista de probabilidades de idiomas posibles.
- Para mejorar la precisión, es útil limpiar y pre-procesar el texto, eliminando caracteres especiales y números y asegurándose de que el texto contenga suficientes palabras o caracteres.

Ejemplo de `detect_langs()`:

```
from langdetect import detect_langs

texto = "Nunca más, brother"
idiomas = detect_langs(texto)
print(idiomas) # Imprime una lista de objetos Language con la probabilidad de cada idioma.

# Imprime por ejemplo:
# [es:0.8571413510438524, en:0.14285747221775852]
```

Otra opción es usar `fasttext-langdetect` (<https://github.com/zafercavdar/fasttext-langdetect>) que es una herramienta basada en `FastText`, librería desarrollada por Facebook AI Research, para la detección eficiente de idiomas. Recordemos que FastText es particularmente útil para tareas de procesamiento de lenguaje natural y es capaz de generar representaciones vectoriales (embeddings) de palabras y frases.

FastText utiliza modelos de aprendizaje profundo para aprender representaciones de palabras y documentos como vectores. Para la detección de idiomas, se entrena un modelo de clasificación supervisada utilizando textos etiquetados con su respectivo idioma. FastText es conocido por su capacidad para generar representaciones de subpalabras. Esto permite que `fasttext-langdetect` sea efectivo incluso con palabras que no se vieron durante el entrenamiento y es especialmente útil para idiomas con mucha morfología, como el turco o el finlandés. Cuando se le presenta un texto para detectar su idioma, `fasttext-langdetect` transforma el texto en un vector utilizando el modelo FastText y luego clasifica este vector en uno de los idiomas que el modelo ha aprendido.

```
# !pip install fasttext-langdetect

from ftlangdetect import detect

# Ejemplo en Alemán
result = detect(text="Ich liebe die Natur und das Reisen.", low_memory=False)
print(result)

# Ejemplo en Francés
result = detect(text="J'aime lire des livres et écouter de la musique.", low_memory=True)
print(result)

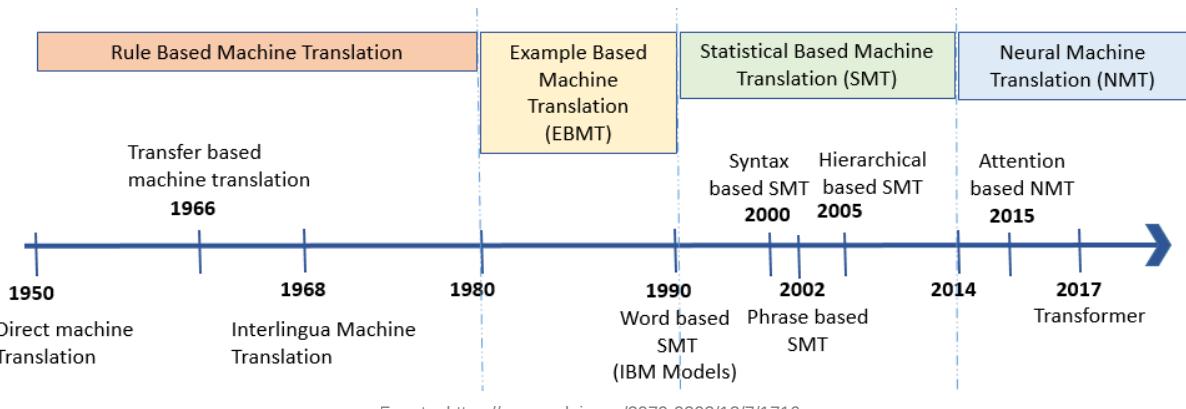
# Imprime:
# {'lang': 'de', 'score': 0.9996129870414734}
# {'lang': 'fr', 'score': 0.992135226726532}
```

FastText generalmente proporciona resultados precisos y robustos, incluso con textos cortos y en diferentes dialectos y formas morfológicas. Además es rápido y eficiente en comparación con otros métodos de detección de idiomas, lo que lo hace útil para aplicaciones en tiempo real.

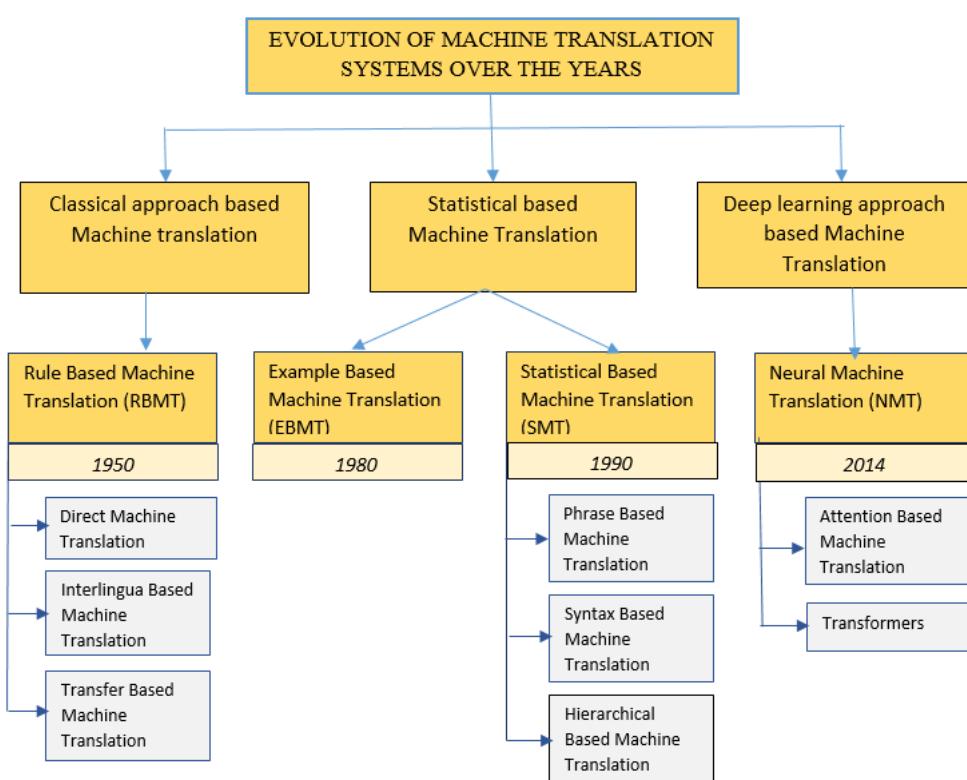
Otras opciones para detección de lenguajes, son las librerías <https://github.com/saffsd/langid.py> o <https://github.com/CLD2Owners/cld2> (C++). También existen muchas opciones en [Hugging Face](#).

## 10. Traducción de texto (Language Translation)

La traducción automática es un subcampo del procesamiento de lenguaje natural que tiene como objetivo realizar la traducción automatizada de un lenguaje natural a otro. La necesidad de entender, compartir e intercambiar ideas de personas multilingües relacionadas con un tema de interés dio origen al campo de la "Traducción Automática". El concepto de automatizar la traducción de idiomas fue concebido por primera vez en 1933 por [Peter Petrovich Troyanskii](#) quien propuso su visión en la Academia de Ciencias; su trabajo se limitó solo a discusiones preliminares. Más tarde, A.D Booth y Warner Weaver en el año 1946 en la Fundación Rockefeller revivieron la idea de automatizar la tarea de traducción. Desde entonces, se han producido muchos avances en términos de poder de cómputo y metodologías que han mejorado la calidad de la traducción. La evolución de los sistemas de traducción automática a lo largo de los años se puede ver en forma de una línea de tiempo:



Se han adoptado varios enfoques hasta ahora para la tarea de traducción automática, los cuales se categorizan en tres categorías principales como se muestra en la siguiente figura:



Aquí mencionamos las cuatro ramas principales de métodos para la traducción automática:

### **RBMT (Traducción Automática Basada en Reglas):**

La traducción automática basada en reglas (RBMT; "Enfoque Clásico" de MT) son sistemas de traducción automática basados en información lingüística sobre los idiomas de origen y destino, obtenida básicamente de diccionarios y gramáticas (unilingües, bilingües o multilingües) que cubren las principales regularidades semánticas, morfológicas y sintácticas de cada idioma respectivamente. Teniendo oraciones de entrada (en algún idioma de origen), un sistema RBMT las genera a oraciones de salida (en algún idioma de destino) basándose en el análisis morfológico, sintáctico y semántico de ambos, el idioma de origen y el idioma de destino involucrados en una tarea de traducción concreta.

### **EBMT (Traducción Automática Basada en Ejemplos)**

La traducción automática basada en ejemplos (EBMT) es un método de traducción automática a menudo caracterizado por su uso de un corpus bilingüe con textos paralelos como su principal base de conocimiento en tiempo de ejecución. Es esencialmente una traducción por analogía y puede ser vista como una implementación de un enfoque de razonamiento basado en casos para el aprendizaje automático.

En la base de la traducción automática basada en ejemplos está la idea de la traducción por analogía. Cuando se aplica al proceso de traducción humana, la idea de que la traducción se realiza por analogía es un rechazo de la idea de que las personas traducen oraciones realizando un análisis lingüístico profundo. En cambio, se basa en la creencia de que las personas traducen descomponiendo primero una oración en ciertas frases, luego traduciendo estas frases y, finalmente, componiendo adecuadamente estos fragmentos en una oración larga. Las traducciones frasales se traducen por analogía a traducciones previas. El principio de la traducción por analogía está codificado en la traducción automática basada en ejemplos a través de las traducciones de ejemplo que se utilizan para entrenar dicho sistema.

### **SMT (Traducción Automática Estadística)**

La traducción automática estadística (SMT) fue un enfoque de traducción automática que superó el enfoque basado en reglas anterior, ya que requería una descripción explícita de cada una de las reglas lingüísticas, lo cual era costoso y a menudo no se generalizaba a otros idiomas. Desde 2003, el enfoque estadístico en sí ha sido gradualmente superado por el enfoque basado en redes neuronales de aprendizaje profundo (NMT).

### **NMT (Traducción Automática Neuronal)**

La **traducción automática neuronal** (NMT por sus siglas en inglés, neural machine translation) es un método de traducción automática que usa redes neuronales, generalmente modelos de aprendizaje profundo como RNN y LSTM, o modelos end-to-end más modernos de tipo transformers.

Aquí vemos un ejemplo de tipo NMT, basado en transformers:

```
from transformers import MarianMTModel, MarianTokenizer

# Define el modelo y el tokenizador
modelo = 'Helsinki-NLP/opus-mt-es-en'
tokenizer = MarianTokenizer.from_pretrained(modelo)
model = MarianMTModel.from_pretrained(modelo)

# Define el texto en español que quieras traducir al inglés
texto_español = "Me gusta aprender procesamiento de lenguaje natural."

# Tokeniza el texto y genera la traducción
inputs = tokenizer(texto_español, return_tensors="pt")
outputs = model.generate(**inputs)

# Decodifica y muestra la traducción
texto_ingles = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(texto_ingles) # Salida: I like to learn new technologies.

# Imprime:
# I like to learn natural language processing.
```

Del mismo modo, tenemos disponible el modelo `Helsinki-NLP/opus-mt-en-es` para la traducción inversa, de inglés a español.

### **Librerías para acceder a servicios**

Existen diversos servicios con modelos de traducción, que están disponibles por medio de APIs. Una opción para acceder a ellos, es utilizar la librería `deep-translator` (<https://github.com/nidhaloff/deep-translator>), que soporta diversas APIs de libre acceso para la traducción de texto. Vamos algunos ejemplos:

#### **Google Translator**

El famoso traductor de Google, es uno de los servicios que se encuentra disponible para ser utilizado desde la librería:

```
from deep_translator import GoogleTranslator
translated = GoogleTranslator(source='auto', target='de').translate('I want to translate this text')
print(translated)
```

#### **PONS**

Es uno de los principales editores de idiomas de Alemania y es famoso por traducir palabras individuales o frases pequeñas. Puede proporcionar sinónimos y sugerencias también:

```
from deep_translator import PonsTranslator
translated_word = PonsTranslator(source='english', target='spanish').translate('good', return_all=False)
print(translated_word)
```

```
translated_word = PonsTranslator(source='english', target='spanish').translate('good', return_all=True)
print(translated_word)
```

### Linguee

Linguee es un servicio web que proporciona un diccionario en línea multilingüe gratuito. A diferencia de servicios similares, Linguee incorpora un motor de búsqueda que proporciona acceso a grandes cantidades de pares de oraciones similares, procedentes de documentos publicados en internet.

```
from deep_translator import LingueeTranslator
translated_word = LingueeTranslator(source='english', target='french').translate('good')
print(translated_word)
```

### MyMemory

El traductor mymemory es la Memoria de Traducción más grande del mundo y es 100% gratuito para usar. Ha sido creado recopilando Memorias de Traducción de la Unión Europea, las Naciones Unidas y alineando los mejores sitios web multilingües específicos de dominio.

```
from deep_translator import MyMemoryTranslator
translated = MyMemoryTranslator(source='en', target='zh').translate(text='cute')
print(translated)
```

Otra librería que soporta múltiples traductores es [translators](https://github.com/uliontse/translators) (<https://github.com/uliontse/translators>). Tiene implementado [más de 30 servicios de traducción](#) y una [cobertura de idiomas](#) enorme. Ejemplo:

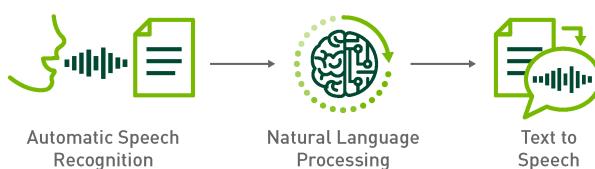
```
import translators as ts

q_text = '最长的路是从迈出第一步开始的。'

print(ts.translate_text(q_text, translator='google', to_language='es'))
```

## 11. Síntesis de voz y TTS (Text-to-speech)

La [síntesis de voz](#), también conocida como Text-to-Speech (TTS), es una tecnología que convierte texto escrito en voz hablada. Esta tecnología es fundamental en diversas aplicaciones como asistentes virtuales, sistemas de navegación, lectores de pantalla para personas con discapacidades visuales, y más.



El TTS (Text-To\_Speech) y el ASR (Automatic Speech Recognition) son componentes claves de los sistemas conversacionales.

### Proceso de Síntesis de Voz:

La síntesis de voz generalmente involucra los siguientes pasos:

#### 1. Preprocesamiento de Texto:

- El texto ingresado se procesa para convertirlo en una forma estructurada, identificando palabras, sílabas, puntuación, etc.
- Se realiza la conversión de texto a fonemas, que son las unidades sonoras del lenguaje.

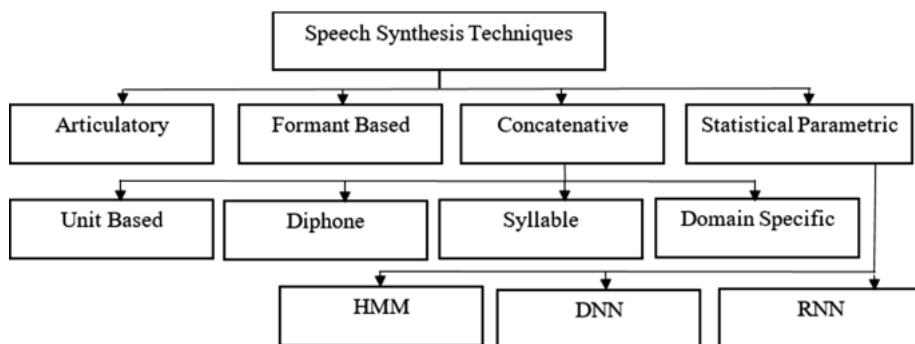
## 2. Generación de Sonido:

- Los fonemas se convierten en sonido utilizando técnicas de síntesis de voz.
- Existen diferentes métodos de síntesis de voz, como la síntesis por concatenación y la síntesis paramétrica.

## 3. Post-procesamiento:

- El sonido generado se procesa para mejorar su calidad y naturalidad, ajustando tono, ritmo, velocidad, etc.

Existen diversos métodos de generación de voz, que se han ido desarrollando desde hace varias décadas:



<https://link.springer.com/article/10.1007/s10462-022-10315-0>

Principalmente destacamos los siguientes métodos de Síntesis de Voz:

### 1. Síntesis por Concatenación:

- Utiliza grabaciones de voz humana segmentadas en unidades sonoras pequeñas.
- Concatena estas unidades para generar voz sintetizada.

### 2. Síntesis Paramétrica:

- Utiliza modelos matemáticos para generar voz sintetizada.
- Permite un mayor control sobre las características de la voz, pero puede sonar menos natural.

### 3. Síntesis de Voz basada en Aprendizaje Profundo:

- Utiliza redes neuronales para modelar y generar voz.
- Puede producir voz muy natural y es capaz de imitar diferentes estilos y emociones.



Los modelos de síntesis de voz más avanzados, incorporan emocionalidad en la expresión de la voz, para lograr un discurso más realista y menos formal. (<https://arxiv.org/pdf/2210.03538.pdf>)

En Python, podemos usar el servicio de Google Text-to-Speech (gTTS), el cual es un servicio de síntesis de voz proporcionado por Google que convierte texto en voz hablada. Este servicio utiliza tecnologías avanzadas de aprendizaje profundo para sintetizar voz que suena natural. Podemos utilizar el servicio a través de la librería `gTTS` (<https://github.com/pndurette/gTTS>):

```
# !pip install gTTS
from gtts import gTTS
import os

# Texto que quieras convertir a voz
texto = "Hola, ¿cómo estás?"

# Crear un objeto gTTS
tts = gTTS(text=texto, lang='es') # lang='es' para español

# Guardar el archivo de audio
tts.save("saludo.mp3")

# Reproducir el archivo de audio
# os.system("start saludo.mp3")
```



Google Text-to-Speech requiere una conexión a Internet, ya que el procesamiento se realiza en los servidores de Google

Para realizar síntesis de voz de manera offline, es posible usar otras alternativas en Python como <https://github.com/nateshmbhat/pyttsx3> y <https://github.com/coqui-ai/TTS>

Otro recurso interesante sobre síntesis de voz y TTS es este repositorio <https://github.com/mozilla/TTS> de Mozilla

## 12. Modelado de Tópicos (Topic Modeling)

El modelado de tópicos ([Topic Modeling](#)) en el Procesamiento del Lenguaje Natural (NLP) es una técnica que se utiliza para descubrir los tópicos o temas subyacentes en un conjunto de documentos. Es una forma de agrupación de textos que permite identificar patrones semánticos en los datos, agrupando palabras y expresiones que aparecen juntas con frecuencia en diferentes documentos.

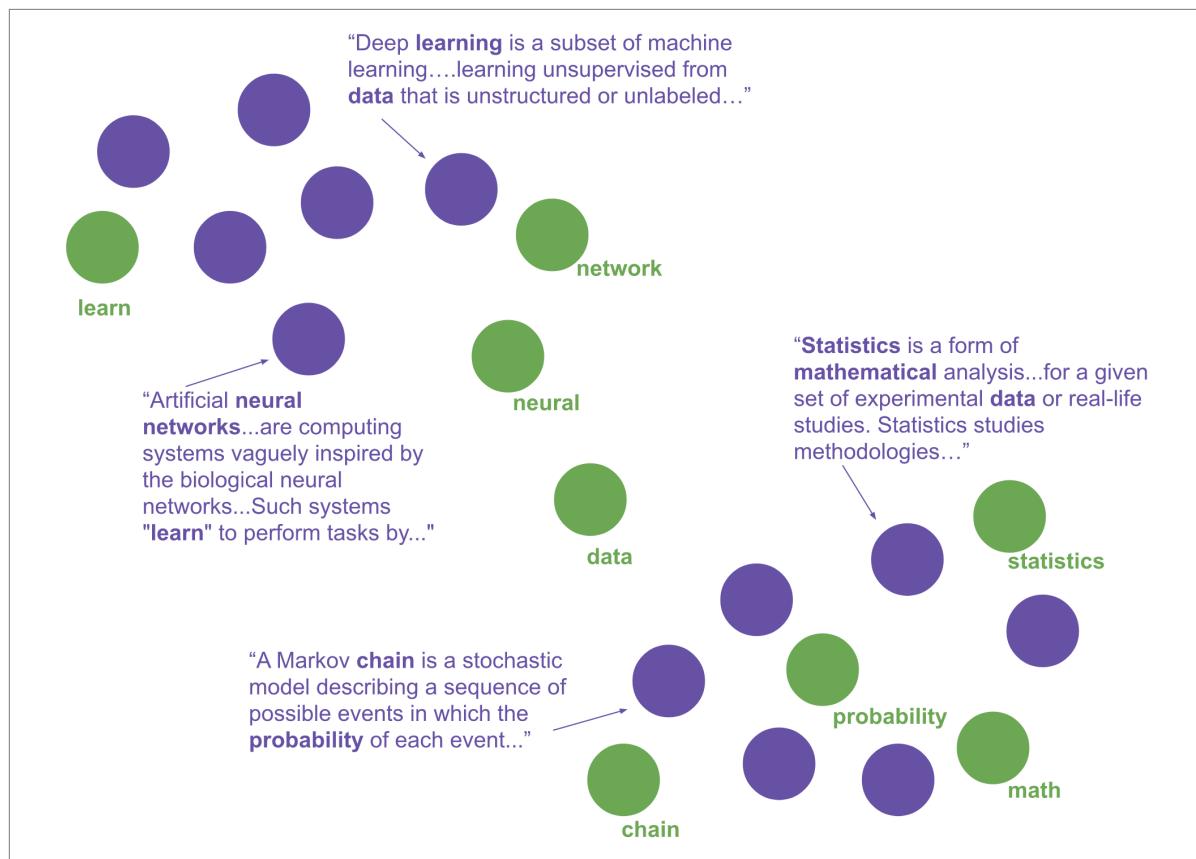
El modelado de tópicos ayuda a descubrir los tópicos o temas comunes en un conjunto de documentos sin tener que etiquetar manualmente los datos. Por ejemplo, en un conjunto de noticias, los tópicos podrían incluir política, deportes, economía, entre otros. También, al representar documentos como una mezcla de tópicos, se reduce la dimensionalidad de los datos, lo que puede ser útil para otras tareas de NLP como clasificación, agrupación o recomendación.

Algunas técnicas comunes de modelado de tópicos incluyen [Latent Dirichlet Allocation \(LDA\)](#), [Non-negative Matrix Factorization \(NMF\)](#), y recientemente modelos basados en deep learning como LDA2Vec (<https://github.com/cemmoody/lda2vec>) y Top2Vec (<https://github.com/ddangelov/Top2Vec>).

Existen herramientas de visualización como pyLDAvis (<https://github.com/bmabey/pyLDAvis>), que permiten explorar los tópicos y su distribución en un conjunto de datos de una manera visual e interactiva, facilitando la interpretación y el análisis.

### Top2Vec (<https://arxiv.org/pdf/2008.09470.pdf>)

Top2Vec, una técnica para el modelado de tópicos que descubre estructuras semánticas latentes en un gran conjunto de documentos. A diferencia de métodos tradicionales como LDA y PLSA, Top2Vec no requiere listas de palabras de parada (stopwords), derivación (stemming) o lematización, y encuentra automáticamente el número de tópicos. Utiliza una incrustación semántica conjunta de documentos y palabras para hallar vectores de tópicos. Los vectores resultantes están incrustados conjuntamente, y la distancia entre ellos representa la similitud semántica. Los experimentos muestran que Top2Vec identifica tópicos más informativos y representativos en comparación con modelos generativos probabilísticos.

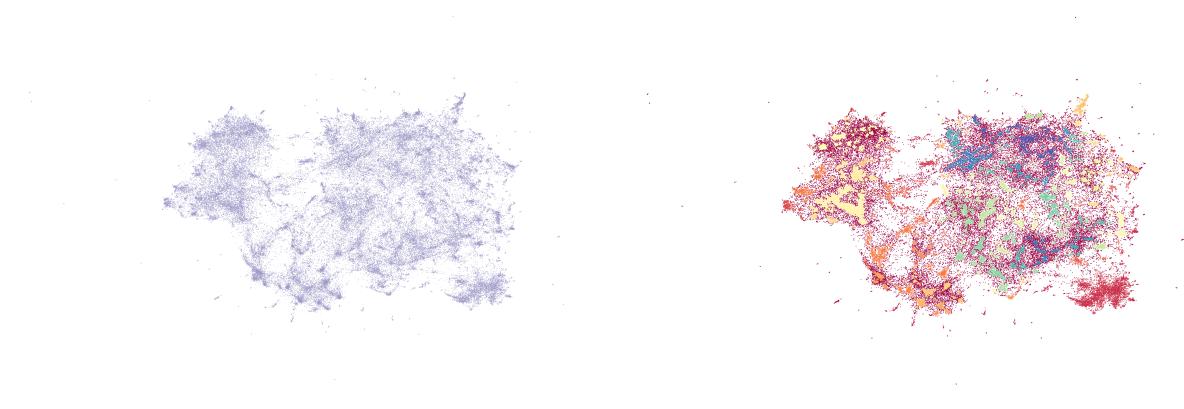


### Reducción de dimensionalidad

Después de tener vectores para cada documento, el siguiente paso natural sería dividirlos en grupos utilizando un algoritmo de agrupación. Sin embargo, los vectores generados en el primer paso pueden tener hasta 512 componentes, dependiendo del modelo de incrustación que se haya utilizado. Por esta razón, tiene sentido realizar algún tipo de algoritmo de reducción de dimensionalidad para reducir el número de dimensiones en los datos. Top2Vec utiliza un algoritmo llamado UMAP (Aproximación y Proyección de Variedad Uniforme) para generar vectores de embeddings de menor dimensión para cada documento.

### Agrupar los vectores

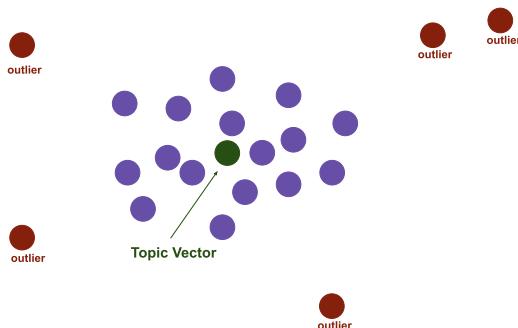
Top2Vec utiliza HDBSCAN, un algoritmo de agrupación basado en densidad jerárquica, para encontrar áreas densas de documentos. HDBSCAN es básicamente una extensión del algoritmo DBSCAN que lo convierte en un algoritmo de agrupación jerárquica. Utilizar HDBSCAN para modelado de temas tiene sentido porque los temas más grandes pueden consistir en varios subtemas.



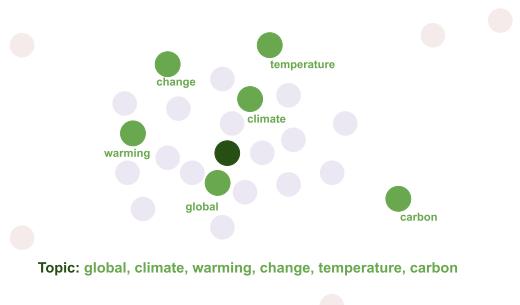
Se crean embeddings de los documentos de una menor dimensión, usando UMAP. Los vectores de documentos en un espacio de alta dimensión son muy dispersos; la reducción de dimensión ayuda a encontrar áreas densas. Cada punto es un vector de documento.

Se buscan las áreas densas de documentos usando HDBSCAN. Las áreas coloreadas son las áreas densas de documentos. Los puntos rojos son valores atípicos que no pertenecen a un grupo específico.

Para cada área densa, se calcula el centroide de los vectores de documentos en la dimensión original; este es el vector del tema. Los puntos rojos son documentos atípicos y no se utilizan para calcular el vector del tema. Los puntos morados son los vectores de documentos que pertenecen a un área densa, a partir de la cual se calcula el vector del tema.



Luego se buscan los n vectores de palabras más cercanos al vector de tema resultante. Los vectores de palabras más cercanos, en orden de proximidad, se convierten en las palabras del tema:



Veamos como aplicar Top2Vec en la práctica. Primero, al instalar la librería Top2Vec, debemos optar por algún método de embeddings, que pueden ser alguno de los siguientes:

- [Doc2Vec](#)
- [Universal Sentence Encoder](#)
- [BERT Sentence Transformer](#):

Según el caso, utilizaremos alguno de estos comandos:

```
# Doc2Vec
pip install top2vec
# BERT Sentence Transformer
pip install top2vec[sentence_transformers]
# Universal Sentence Encoder
pip install top2vec[sentence_encoders]
```

Veamos ahora un ejemplo de como aplicar `top2vec[sentence_encoders]`

Utilizaremos el dataset 20 Newsgroups (<https://www.kaggle.com/datasets/crawford/20-newsgroups>) que cuenta con más de 18.000 posts de mensajes en 20 grupos de noticias. Este dataset lo podemos usar fácilmente gracias a `sklearn.datasets.fetch_20newsgroups`.

```
# !pip install top2vec[sentence_encoders]

import numpy as np
import pandas as pd
from top2vec import Top2Vec
from sklearn.datasets import fetch_20newsgroups

# Cargar el conjunto de datos 20newsgroups, eliminando encabezados, pies de página y citas
newsgroups = fetch_20newsgroups(subset='all', remove=('headers', 'footers', 'quotes'))

# Crear un DataFrame con los datos
articles_df = pd.DataFrame({'content': newsgroups.data})

# Crear un modelo Top2Vec usando el modelo de embedding 'universal-sentence-encoder'.
# Este paso puede demorar algunos minutos, según el tamaño de nuestro corpus.
model = Top2Vec(articles_df['content'].values, embedding_model='universal-sentence-encoder')

# Buscar tópicos relacionados con la palabra clave "medicine", solicitando los 3 tópicos más relevantes
topic_words, word_scores, topic_nums = model.search_topics(keywords=["medicine"], num_topics=3)
```

En el ejemplo, creamos un DataFrame de pandas llamado `articles_df` que contiene el contenido de los documentos de noticias. Luego se crea un modelo, utilizando el contenido de los documentos. Para ello, se utiliza el modelo de embeddings `universal-sentence-encoder`, que es un modelo pre-entrenado para convertir oraciones en vectores numéricos. Este paso puede llevar tiempo, ya que el modelo procesa y aprende de todo el conjunto de datos. En caso que quisieramos trabajar en otros idiomas diferentes al inglés, podemos usar `embedding_model=universal-sentence-encoder-multilingual`

Luego se utiliza el modelo Top2Vec para buscar tópicos relacionados con las palabras clave "medicine", "doctors", "surgery". Se solicitan los 3 tópicos más relevantes relacionados con esta palabra clave. Como resultado, se obtienen las palabras asociadas a cada tópico, las puntuaciones de esas palabras, las puntuaciones de los tópicos y los números de tópico.

Ahora que tenemos nuestro modelo, podemos realizar búsquedas por palabras clave con `model.search_topics`:

```
# Buscar tópicos relacionados con la palabra clave "medicine", "doctors", "surgery", solicitando los 3 tópicos más relevantes
topic_words, word_scores, topic_nums = model.search_topics(keywords=["medicine", "doctors", "surgery"], num_topics=3)

# Mostrar las palabras clave y las puntuaciones de los tópicos encontrados
for i, topic_num in enumerate(topic_nums):
    print(f"Tópico {i + 1}:")
    print(f"Palabras clave: {', '.join(topic_words[i])}")
    print(f"Puntuación del tópico: {topic_scores[i]}\n")
```

Y obtendremos lo siguiente:

```
Tópico 1:
Palabras clave: diagnosed, symptoms, diagnosis, severe, vax, treatments, immune, dr, doctors, surgery, candida, cure, patients, lyme, t
Puntuación del tópico: 0.27713936810628437

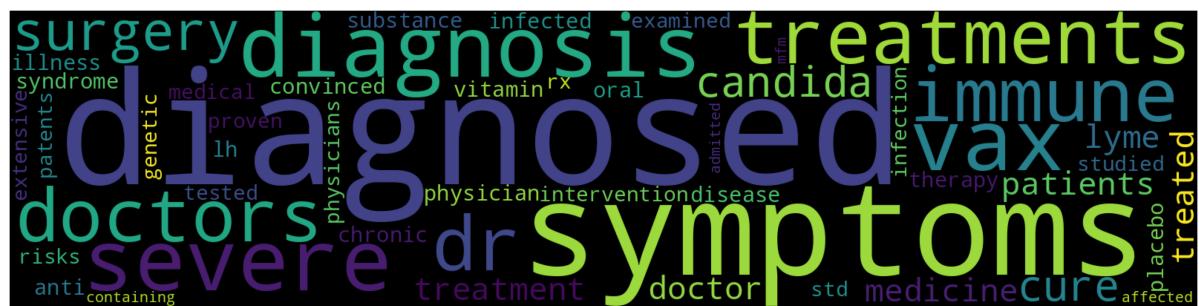
Tópico 2:
Palabras clave: um, ci, oo, ye, ll, hmm, un, uh, y_, wt, on, uu, actually, an, eh, way, er, des, se, not, has, huh, of, ya, so, in, it,
Puntuación del tópico: 0.26305345157739546

Tópico 3:
Palabras clave: drugs, drug, tobacco, crack, substance, feds, dro, atf, intervention, medicine, vice, reform, libertarian, libertarians
Puntuación del tópico: 0.17469598439399736
```

Una vez que tenemos las palabras clave, también las podemos graficar en una nube de palabras:

```
print(model.topic_words[1])
model.generate_topic_wordcloud(1)
```

## Topic 1



También podemos buscar los documentos que pertenecen a un determinado tópico:

```
# Mostrar algunos documentos relacionados con el primer tópico encontrado
documents, document_scores, document_ids = model.search_documents_by_topic(topic_num=topic_nums[0], num_docs=2)
for i, doc in enumerate(documents):
    print(f"Documento {i + 1}:")
    print(f"Contenido: {doc}")
    print(f"Puntuación del documento: {document_scores[i]}\n")
```