

# Unidad 3 - Procesamiento del Lenguaje

UNR - TUIA - Procesamiento de Lenguaje Natural

Docente teoría: Juan Pablo Manson - [jpmanson@gmail.com](mailto:jpmanson@gmail.com) - [LinkedIN](#)

## 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.
5. **Similitud de Jaro-Winkler**: Una variante de la distancia de Jaro que da más peso a las coincidencias al inicio de las cadenas. Es particularmente útil para comparar cadenas cortas como nombres o códigos.

## 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.",
    "El firmamento se ve azulado y sin nubes a la vista.",
    "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 otras
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.

```
# Importar las bibliotecas necesarias
from sklearn.metrics import jaccard_score
from sklearn.feature_extraction.text import CountVectorizer
```

```

# Lista de pares de frases a comparar
# Incluye ejemplos originales y nuevos ejemplos con las mismas palabras pero diferentes significados
frases = [
    ("El gato negro salta sobre el sofá.", ".") # Significado similar
    ("Los niños juegan en el parque con una pelota.", "Los niños corren en el parque")
    ("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")
    # Nuevos ejemplos con las mismas palabras, diferente orden/significado
    ("Juan ama a María.", "María ama a Juan."), # Mismas palabras, diferente significado
    ("Solo Juan come pescado.", "Juan solo come pescado.") # Mismas palabras, diferente significado
]

# Iterar a través de cada par de frases
for frase_a, frase_b in frases:
    # Inicializar CountVectorizer para convertir texto a vectores de conteo de palabras
    # binary=True asegura presencia/ausencia (1/0) en lugar de conteos
    vectorizer = CountVectorizer(binary=True)

    # Ajustar el vectorizador a las frases y transformarlas en vectores
    X = vectorizer.fit_transform([frase_a, frase_b])

    # Extraer los arrays de vectores para cada frase
    vector_1 = X[0].toarray()[0]
    vector_2 = X[1].toarray()[0]

    # Calcular el puntaje de similitud de Jaccard
    # El promedio 'micro' es adecuado aquí ya que comparamos dos vectores de dimensiones diferentes
    similarity = jaccard_score(vector_1, vector_2, average='micro')

    # Imprimir los resultados para el par actual
    print(f"Similitud Jaccard entre\n\"{frase_a}\"\ny\n\"{frase_b}\"\nes: {similarity:.2f}")

```

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

Jaccard Similarity between

"Juan ama a María."

and

"María ama a Juan."

is: 1.0

Jaccard Similarity between

"Solo Juan come pescado."

and

"Juan solo come pescado."

is: 1.0

## 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:
# 0.666666666666666
```

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
```

Otra forma de aplicar Levenshtein es con la librería `thefuzz` (<https://github.com/seatgeek/thefuzz>). Aquí lo aplicaremos para hacer

búsquedas flexibles en una columna de un Dataframe de Pandas:

```
# !pip install thefuzz python-Levenshtein
import pandas as pd
from thefuzz import process

# Crear un DataFrame de ejemplo
df = pd.DataFrame({
    'Nombre': ['Juan Pérez', 'María García', 'Pedro Rodríguez', 'Ana Martínez',
    'Luis González'],
    'Edad': [30, 25, 35, 28, 40]
})

# Función para realizar fuzzy search
def fuzzy_search(df, column, query, limit=2, threshold=80):
    return process.extractBests(query, df[column], limit=limit, score_cutoff=threshold)

# Ejemplo de uso
query = 'Juan Peres'
resultados = fuzzy_search(df, 'Nombre', query)

print(f"Resultados para '{query}':")
for resultado in resultados:
    nombre, score, _ = resultado
    print(f"Nombre: {nombre}, Puntuación: {score}")

# Ejemplo adicional con una búsqueda más amplia
query_amplio = 'Garcia'
resultados_amplios = fuzzy_search(df, 'Nombre', query_amplio, limit=3, threshold=70)

print(f"\nResultados para '{query_amplio}' (búsqueda más amplia):")
for resultado in resultados_amplios:
    nombre, score, _ = resultado
    print(f"Nombre: {nombre}, Puntuación: {score}")
```

La salida será:

Resultados para 'Juan Peres':  
Nombre: Juan Pérez, Puntuación: 84

Resultados para 'Garcia' (búsqueda más amplia):  
Nombre: María García, Puntuación: 82

La función `fuzzy_search` realiza una búsqueda difusa en una columna específica de un DataFrame de Pandas. Vamos a desglosar sus componentes:

#### 1. Parámetros de la función:

- `df`: El DataFrame de Pandas en el que se realizará la búsqueda.
- `column`: El nombre de la columna del DataFrame donde se buscará.
- `query`: La cadena de texto que se quiere buscar.
- `limit`: El número máximo de resultados a devolver (por defecto 2).
- `threshold`: La puntuación mínima de similitud para considerar una coincidencia (por defecto 80).

#### 2. Cuerpo de la función:

La función utiliza

`process.extractBests` de la biblioteca `thefuzz`. Esta función realiza la búsqueda difusa y devuelve los mejores resultados.

#### 3. `process.extractBests`:

- Primer argumento `query`: Es la cadena que estamos buscando.
- Segundo argumento `df[column]`: Es la serie de Pandas que contiene todos los valores de la columna especificada.
- `limit`: Especifica cuántos resultados queremos obtener como máximo.
- `score_cutoff`: Es el umbral mínimo de puntuación para considerar una coincidencia.

#### 4. Funcionamiento:

- La función compara la `query` con cada elemento en `df[column]`.
- Utiliza algoritmos de comparación de cadenas (por defecto, la distancia de Levenshtein) para calcular una puntuación de similitud entre 0 y 100.
- Devuelve una lista de tuplas. Cada tupla contiene:

- El valor encontrado en la columna.
- La puntuación de similitud.
- El índice del elemento en el DataFrame original.

## 5. Resultado:

- La función devuelve directamente el resultado de `process.extractBests`, que es una lista de las mejores coincidencias que cumplen con los criterios de `limit` y `threshold`.

Esta función es muy útil porque:

- Permite buscar coincidencias aproximadas, tolerando errores de escritura o variaciones en los nombres.
- Es flexible: puedes ajustar la precisión de la búsqueda cambiando el `threshold`.
- Puedes controlar cuántos resultados quieres obtener con el parámetro `limit`.

Por ejemplo, si buscamos "Juan Peres" en una columna que contiene "Juan Pérez", probablemente obtendremos una coincidencia con una puntuación alta, a pesar del error ortográfico en el apellido.



Es importante destacar que este tipo de búsquedas flexibles (fuzzy search) se centran principalmente en la similitud ortográfica y estructural de las cadenas de texto, no en su significado semántico.

## Similitud de Jaro-Winkler

La similitud de Jaro-Winkler es una métrica usada para medir la similitud entre dos cadenas de caracteres, que es útil en la corrección de errores de entrada, la comparación de nombres y otros tipos de coincidencia de texto donde las cadenas podrían tener errores tipográficos o ser variantes fonéticas de la misma palabra. La similitud de Jaro mide cuán similar es una cadena a otra basándose en la cantidad y el orden de los caracteres comunes, además de las transposiciones (caracteres en orden incorrecto) entre ellas. Se define como:

$$J = \frac{1}{3} \left( \frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m - t}{m} \right)$$

$$J_w = J + l \cdot p \cdot (1 - J)$$

Donde:

- $J$  es la puntuación de similitud de Jaro.
- $l$  es la longitud del prefijo común en el inicio de las cadenas (hasta un máximo de 4 caracteres).
- $p$  es un factor de escalamiento (típicamente  $p=0.1$ ).
- $|s_1|$  y  $|s_2|$  son las longitudes de las cadenas.
- $m$  es el número de caracteres coincidentes.
- $t$  es la mitad del número de transposiciones.

La similitud de Jaro-Winkler es ampliamente usada en sistemas de bases de datos y software de entrada de datos para reducir errores por mala digitación o mal entendido fonético.

La principal ventaja de la similitud de Jaro-Winkler es su capacidad para manejar errores pequeños en la entrada de datos, lo que la hace robusta en entornos donde los errores de entrada son comunes. Sin embargo, tiene algunas limitaciones, ya que no maneja bien las grandes diferencias en la longitud de las cadenas y puede ser computacionalmente intensiva en grandes bases de datos sin optimizaciones adecuadas.

Veamos un ejemplo para encontrar coincidencias de nombres:

```
import jellyfish # Biblioteca que implementa Jaro-Winkler
from typing import List, Tuple

def find_name_matches(name: str, name_list: List[str], threshold: float = 0.85)
    """
    Encuentra coincidencias de nombres usando la similitud de Jaro-Winkler.

    :param name: El nombre a buscar
    :param name_list: Lista de nombres en la que buscar
    :param threshold: Umbral de similitud (por defecto 0.85)
    :return: Lista de tuplas con nombres coincidentes y sus puntuaciones de similitud
    """

    matches = []
    for candidate in name_list:
        similarity = jellyfish.jaro_winkler_similarity(name.lower(), candidate.lower())
        if similarity >= threshold:
            matches.append((name, candidate, similarity))

    return matches
```

```

matches.append((candidate, similarity))

return sorted(matches, key=lambda x: x[1], reverse=True)

# Lista de ejemplo de nombres
name_database = [
    "John Smith", "Jon Smith", "John Smyth", "Jane Smith",
    "Mary Johnson", "Marie Jonson", "Mark Johnson", "Roberto Sánchez Ocam",
    "William Brown", "Bill Brown", "Will Brown",
    "Elizabeth Taylor", "Elisabeth Taylor", "Liz Taylor",
    "Michael Williams", "Mike Williams", "Micheal Williams",
    "Sarah Davis", "Sara Davies", "Sarah Davies"
]

# Ejemplos de uso
test_names = ["Jon Smyth", "Mary Jonson", "Bill Brwn", "Elisabeth Tylor"]

for test_name in test_names:
    print(f"\nBuscando coincidencias para: {test_name}")
    matches = find_name_matches(test_name, name_database)
    if matches:
        for match, score in matches:
            print(f" - {match} (Similitud: {score:.4f})")
    else:
        print(" No se encontraron coincidencias con suficiente similitud.")

# Ejemplo de cómo ajustar el umbral
print("\nBuscando 'Mike Williams' con un umbral más bajo:")
low_threshold_matches = find_name_matches("Mike Williams", name_database, threshold=0.8)
for match, score in low_threshold_matches:
    print(f" - {match} (Similitud: {score:.4f})")

```

La salida será:

```

Buscando coincidencias para: Jon Smyth
- John Smyth      (Similitud: 0.9733)
- Jon Smith       (Similitud: 0.9556)
- John Smith      (Similitud: 0.9170)

```

Buscando coincidencias para: Mary Jonson

- Mary Johnson (Similitud: 0.9833)
- Marie Jonson (Similitud: 0.9399)
- Mark Johnson (Similitud: 0.9399)

Buscando coincidencias para: Bill Brwn

- Bill Brown (Similitud: 0.9800)
- Will Brown (Similitud: 0.8963)

Buscando coincidencias para: Elisabeth Tylor

- Elisabeth Taylor (Similitud: 0.9875)
- Elizabeth Taylor (Similitud: 0.9553)

Buscando 'Mike Williams' con un umbral más bajo:

- Mike Williams (Similitud: 1.0000)
- Micheal Williams (Similitud: 0.8462)
- Michael Williams (Similitud: 0.8239)

## 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:

```
#!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
Autoridades europeas indicaron que el verano boreal de 2023 fue el más cálido
Las temperaturas medias mundiales durante los tres meses del verano boreal,
Nuestro clima está implosionando más rápido de lo que podemos hacer frente
Canículas, sequías, inundaciones o incendios azotaron durante ese verano bo
"""

# 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 range
Entidad: Nuestro, Etiqueta: PER, Explicación: Named person or family.
Entidad: Canículas, Etiqueta: LOC, Explicación: Non-GPE locations, mountain range
Entidad: Asia, Etiqueta: LOC, Explicación: Non-GPE locations, mountain range
Entidad: Europa, Etiqueta: LOC, Explicación: Non-GPE locations, mountain range
Entidad: América del Norte, Etiqueta: LOC, Explicación: Non-GPE locations, mountain range

```

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: Países, 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.

En este ejemplo de código veremos como usar la librería `gliner` (<https://github.com/urchade/GLiNER>) para implementar una tarea de reconocimiento de entidades nombradas (NER) en un texto. La librería `gliner` es útil en el procesamiento de lenguaje natural y permite identificar y categorizar

entidades específicas dentro de un texto, como nombres de personas, localizaciones, fechas, actores y personajes ficticios:

```
# !pip install gliner
# Importa la clase GLiNER desde la biblioteca gliner
from gliner import GLiNER

# Lista de modelos disponibles: https://huggingface.co/urchade

# Carga el modelo preentrenado 'gliner_multi-v2.1' desde Hugging Face
model = GLiNER.from_pretrained("urchade/gliner_multi-v2.1")

# Cambia el modelo a modo de evaluación, esto es útil para desactivar caracteres
model.eval()

# Define el texto a procesar
text = """
Leave the World Behind está aquí y es escalofriante. Protagonizada por Julia
"Siempre me había interesado hacer una película de desastres, y específicamente
La película está basada en la novela superventas de 2020 del mismo nombre
"""

# Lista de etiquetas que el modelo intentará encontrar en el texto
labels = ["person", "book", "location", "date", "actor", "character"]

# Realiza la predicción de entidades en el texto dado, utilizando las etiquetas definidas
entities = model.predict_entities(text, labels, threshold=0.4)

# Imprime cada entidad detectada junto con su etiqueta
for entity in entities:
    print(entity["text"], "⇒", entity["label"])
```

Y el resultado será el siguiente:

```
Julia Roberts ⇒ actor
Mahershala Ali ⇒ actor
Ethan Hawke ⇒ actor
Sam Esmail ⇒ person
```

Mr. Robot ⇒ character  
Long Island ⇒ location  
2020 ⇒ date  
Rumaan Alam ⇒ book  
Tonia Davis ⇒ person  
Daniel M. Stillman ⇒ person  
Nick Krishnamurthy ⇒ person

Más información de Gliner en:

#### Meet the new zero-shot NER architecture



GLiNER is a Named Entity Recognition (NER) model capable of identifying any entity type using a bidirectional transformer encoder...

 <https://blog.knowledgator.com/meet-the-new-zero-shot-ner-architecture-30ffc2cb1ee0>

## 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. **Análisis 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:** Diversos 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 pueden aplicar 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 pe
```

```

# 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

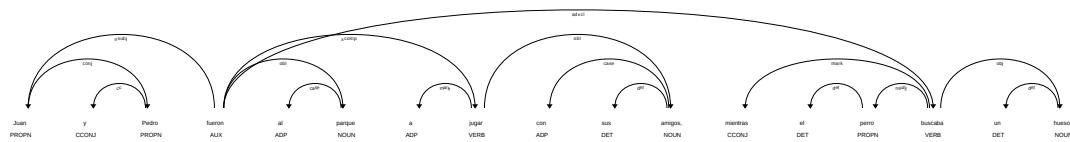
17	hueso	NOUN	noun
18	.	PUNCT	punctuation

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

POS	DESCRIPCIÓN	EJEMPLOS
INTJ	Interjección	¡eh!, ¡ay!, ¡bravo!, ¡hola!
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	sfpkdspxmsa
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:

```
# !pip install stanza
import stanza

# Descargamos el modelo español
stanza.download('es')

# Inicializamos el pipeline de procesamiento español
nlp = stanza.Pipeline('es')

# Texto para analizar
doc = nlp("El autobot de aspecto humanoide, llamado Apollo, está pensado pa

# Navegamos cada oración de nuestro texto
for i, sent in enumerate(doc.sentences):
    print("[Sentence {}]".format(i+1))
```

```

for word in sent.words:
    print("{:12s}\t{:12s}\t{:6s}\t{:d}\t{:12s}".format(
        word.text, word.lemma, word.pos, word.head, word.deprel))
print("")

```

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".
- **advmod:** (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.

### Análisis de contenido usando POS

Otro ejemplo donde aplicar POS, es en el análisis de contenidos. Analicemos los siguientes textos, con estilos de escritura diferentes:

Texto "Noticia":

El gobierno anunció nuevas medidas económicas para controlar la inflación. Según el ministro de economía, se espera que estas regulaciones estabilicen cambiario. Los analistas financieros expresaron opiniones divididas sobre el ir corto plazo. La bolsa de valores reaccionó con leves caídas tras el comunicado.

Texto "Literario":

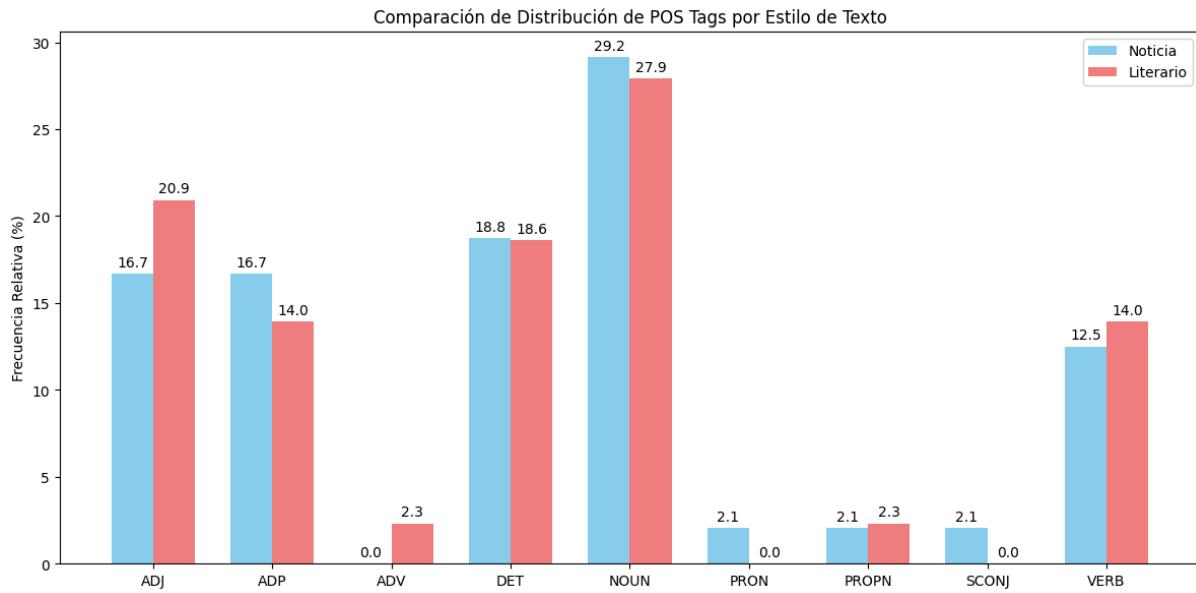
La tarde caía lentamente sobre la ciudad dormida.  
Viejas farolas parpadeaban, arrojando una luz tenue sobre el empedrado húmedo.  
Un gato solitario cruzó la calle en silencio, buscando refugio del frío incipiente.  
El viento susurraba secretos entre los árboles desnudos del parque cercano.

La siguiente aplicación de la herramienta POS, nos permite obtener algunas estadísticas:

Google Colab

🔗 [https://colab.research.google.com/drive/1guY9bIMZLSB  
DKGeqt7xx7pDnHvLef7ZP?usp=sharing](https://colab.research.google.com/drive/1guY9bIMZLSBDKGeqt7xx7pDnHvLef7ZP?usp=sharing)





Del análisis, se pueden sacar algunas conclusiones preliminares, aunque siempre con la **precaución de que los textos de ejemplo son muy cortos**:

- 1. Mayor Uso de Adjetivos (ADJ) en Texto Literario:** La diferencia más notable es en los adjetivos. El texto literario usa un porcentaje considerablemente mayor (20.93%) en comparación con la noticia (16.67%). Esto es esperable, ya que la literatura a menudo se apoya más en la descripción y la evocación de sensaciones, roles que cumplen los adjetivos (ej. "ciudad *dormida*", "luz *tenue*", "empedrado *húmedo*", "gato *solitario*", "frío *incipiente*", "árboles *desnudos*").
- 2. Presencia de Adverbios (ADV) Solo en Texto Literario:** El texto literario es el único que registra adverbios (2.33%), como "lentamente" y "silencio". Los adverbios también contribuyen a la calidad descriptiva o narrativa, modificando verbos, adjetivos u otros adverbios.
- 3. Uso Similar de Sustantivos (NOUN) y Determinantes (DET):** Ambos textos tienen una alta proporción de sustantivos y determinantes, siendo las categorías más frecuentes en ambos casos. La noticia tiene una ligera predominancia de sustantivos, lo cual podría reflejar un enfoque más centrado en entidades, hechos y conceptos ("gobierno", "medidas", "inflación", "ministro", "mercado", "analistas", "bolsa").
- 4. Ligeramente Más Preposiciones (ADP) en Noticias:** La noticia usa un poco más de preposiciones (16.67% vs 13.95%). Esto podría deberse a la necesidad de estructurar información y relaciones entre conceptos

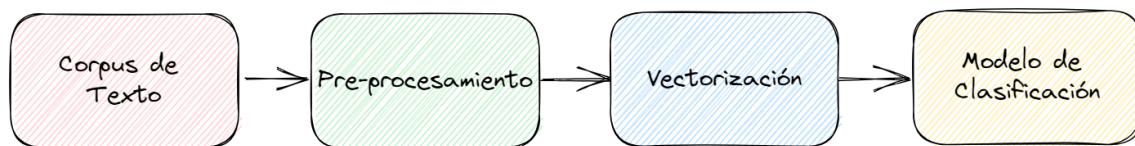
("medidas para controlar", "según el ministro", "opiniones sobre el impacto", "caídas tras el comunicado").

5. **Uso Similar de Verbos (VERB):** La proporción de verbos es bastante parecida en ambos textos.

Basándose en estas muestras, el texto **literario muestra una tendencia hacia un lenguaje más descriptivo y matizado**, evidenciado por el mayor uso de adjetivos y la presencia de adverbios. El texto de **noticia parece ser ligeramente más denso en sustantivos y utiliza más elementos estructurales** como las preposiciones para conectar la información factual.

## 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.

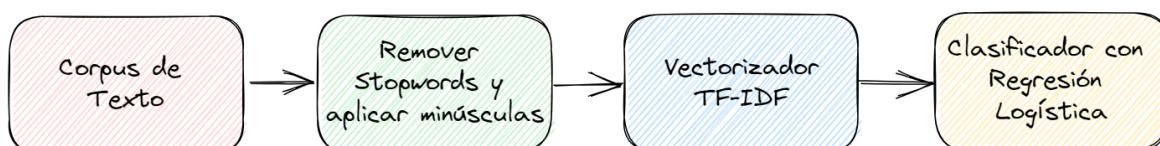
El proceso general para construir un clasificador de texto incluye los siguientes pasos:

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 pre-procesados 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. Como ya hemos visto, 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 de Scikit-Learn 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."))
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 C++."))

# 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 multinomial
modelo_LR = LogisticRegression(max_iter=1000, multi_class='multinomial', so
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
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[etiquetas_predichas[i]]}")

```

Si quisiéramos 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

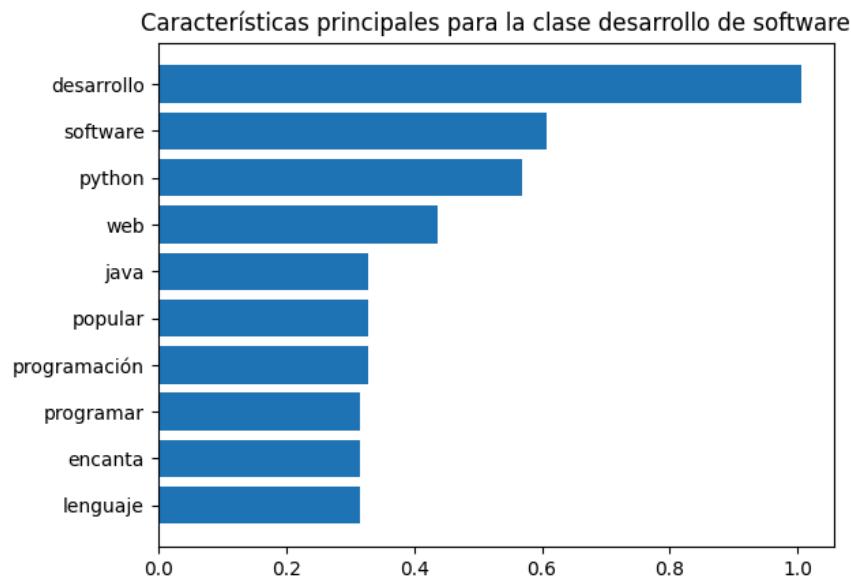
# Obtén los nombres de las características y los coeficientes
feature_names = vectorizer.get_feature_names_out()
coef = modelo_LR.coef_

# 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”:



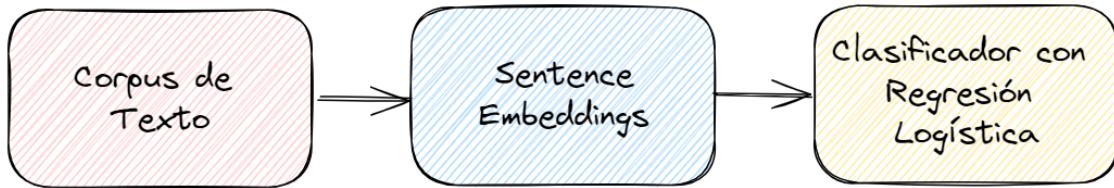
Esta visualización nos permite entender mejor como funciona el clasificador, e incluso realizar ajustes a nuestro dataset de entrenamiento si es necesario, para mejorar el rendimiento.



`vectorizer.get_feature_names_out()` es un método del objeto `TfidfVectorizer` de scikit-learn que devuelve una lista de todas las características (palabras o términos) que el vectorizador ha recolectado del conjunto de datos de entrenamiento. Estas características son las que el vectorizador utiliza para transformar los textos en vectores numéricos.

## Ejemplo con modelo de vectorización semántico

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 `all-mnppnet-base-v2` en reemplazo de TF-IDF. Además hemos quitado la eliminación de Stopwords, ya que prácticamente no nos afectará a la hora de hacer Sentence Embeddings. Nuestro nuevo código, aplicando el modelo semántico será el siguiente:

```

# !pip install transformers sentence_transformers

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

# Cargamos el modelo desde HuggingFace https://huggingface.co/sentence-
model = SentenceTransformer('sentence-transformers/all-mnppnet-base-v2')

labels = [(0, "desarrollo de software"), (1, "videojuegos"), (2, "inteligencia artif
(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((3, "La autenticación de dos factores es una técnica de cibers"))
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 = model.encode(X_train)
X_test_vectorized = model.encode(X_test)

# Creación y entrenamiento del modelo de Regresión Logística Multinomial
modelo_LR = LogisticRegression(max_iter=1000, multi_class='multinomial', solver='saga')
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.",
    "Me encanta la programación en Python."
]

```

```

    "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 = model.encode(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")

```

Como vemos en el resultado, nuestras métricas con este modelo y este dataset, son inmejorables:

```

Precisión Regresión Logística: 1.0
Reporte de clasificación Regresión Logística:
    precision    recall   f1-score   support
    1         1.00     1.00     1.00      3
    2         1.00     1.00     1.00      4
    3         1.00     1.00     1.00      3

    accuracy          1.00      10
macro avg       1.00     1.00     1.00      10
weighted avg    1.00     1.00     1.00      10

```

Texto: 'Quisiera aprender a programar en Rust.'

Clasificación predicha: desarrollo de software

Texto: 'Los videojuegos de realidad virtual son increíbles.'

Clasificación predicha: videojuegos

Texto: 'La inteligencia artificial está avanzando rápidamente.'

Clasificación predicha: inteligencia artificial

Aquí vemos una comparación resumida de los métodos de clasificación que acabamos de crear, de acuerdo a la forma de vectorizar el texto en cada caso:

Característica	TF-IDF	Sentence Embeddings
Representación del texto	Vector disperso basado en frecuencia de palabras	Vector denso que captura significado semántico
Contexto y semántica	No captura contexto o significado semántico	Captura contexto y relaciones semánticas entre palabras
Manejo de sinónimos	No reconoce sinónimos a menos que aparezcan explícitamente	Puede reconocer y relacionar palabras semánticamente similares
Dimensionalidad	Depende del tamaño del vocabulario. Con varios documentos sería muy Alta	Fija (768 para BERT)
Palabras fuera del vocabulario	No puede manejarlas	Puede generar representaciones basadas en contexto y subpalabras
Complejidad computacional	Simple y rápido de calcular	Requiere más recursos para entrenamiento e inferencia
Rendimiento en clasificación	Bueno en tareas simples o datasets pequeños	Mejor en tareas complejas y datasets grandes
Interpretabilidad	Más interpretable, cada dimensión es una palabra	Menos interpretable directamente
Pre-entrenamiento	No requiere	Utiliza modelos de lenguaje pre-entrenados

## 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 clásificador 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
```

Frase: 'The food was delicious, I will definitely come back again.'

Sentimiento: 5 stars, Score: 0.525

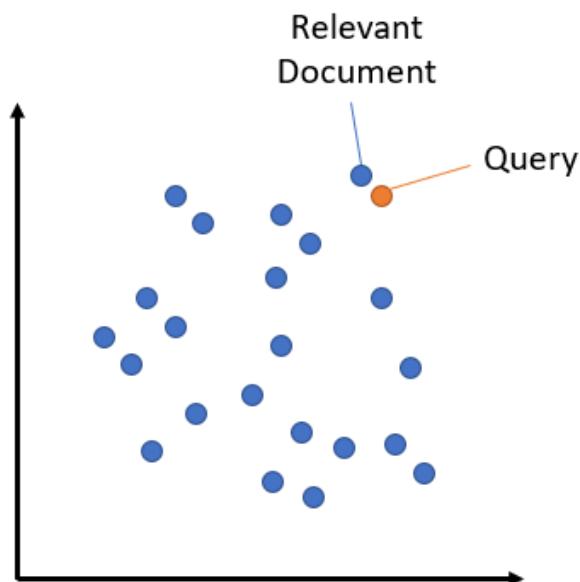
Frase: 'El lugar está un poco descuidado y sucio.'

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 era en 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',
    'El Tiranosaurio Rex vivió hace 68 y 66 millones de años en las últimas etapas',
    'Los dinosaurios se extinguieron de la faz de la tierra, debido al impacto de un asteroide',
    '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 energía solar en energía química que utilizan para crecer'
]

# 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)
    mejor_indice = similitudes.argmax()
```

```
print(f"Consulta: {consultas[i]}")  
print(f"Mejor respuesta (Similitud: {similitudes[mejor_indice]:.4f}): {respues  
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  
  
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  
  
Consulta: ¿De qué depende la vida en el planeta?  
Mejor respuesta (Similitud: 0.6135): La vida en la Tierra depende fundamentalmente  
  
Consulta: ¿Cuál fue el rey de los dinosaurios?  
Mejor respuesta (Similitud: 0.5627): Los dinosaurios se extinguieron de la faz  
  
Consulta: ¿Cuándo y dónde vivieron los Tiranosaurios?  
Mejor respuesta (Similitud: 0.5550): El Tiranosaurio Rex es considerado el rey  
  
Consulta: ¿Cómo se extinguieron los dinosaurios?  
Mejor respuesta (Similitud: 0.7098): Los dinosaurios se extinguieron de la faz
```

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  
for i, incrustacion_consulta in enumerate(incrustaciones_consultas):  
    hits = util.semantic_search(incrustacion_consulta, incrustaciones_respuesta  
    print(f"Consulta: {consultas[i]}")  
    for j, hit in enumerate(hits[0]):
```

```
    print(f"Respuesta {j+1} (Similitud: {hit['score']:.4f}): {respuestas[hit['corp']]}")
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

Respuesta 2 (Similitud: 0.4600): El Tiranosaurio Rex es considerado el rey de

Respuesta 3 (Similitud: 0.4039): El alimento preferido de los conejos son las z



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. Detección de idioma (Language Detection)

En el procesamiento de lenguaje natural (PNL), la detección de idioma consiste en identificar automáticamente el idioma en el que está escrito un texto. Este proceso es fundamental en aplicaciones de PNL, especialmente en sistemas multilingües y plataformas globales donde los textos pueden estar en diversos idiomas. Para realizar esta tarea, se emplean varios métodos, cada uno con enfoques y herramientas específicas:

### 1. Frecuencia de Caracteres y Palabras:

- Estos métodos analizan la frecuencia de caracteres o palabras en un texto y la comparan con patrones característicos de diferentes idiomas.

Un ejemplo destacado es el modelo n-grama, que utiliza secuencias de caracteres o palabras para determinar el idioma. Por ejemplo, el modelo [n-grama de Cavnar y Trenkle \(1994\)](#) es ampliamente utilizado y está implementado en bibliotecas como <https://github.com/Mimino666/langdetect>, que emplea un clasificador Naive Bayes con n-gramas de caracteres para distinguir entre múltiples idiomas (Language Detection).

## 2. Modelos Estadísticos:

- Los modelos estadísticos o de aprendizaje automático se entrena para reconocer estructuras lingüísticas y peculiaridades de distintos idiomas. Un ejemplo común es el clasificador Multinomial Naive Bayes con Bag of Words, que clasifica textos según la frecuencia de palabras. Este enfoque se ha implementado en proyectos como el de Analytics Vidhya, donde se alcanzó una precisión del 97.7% en la detección de idioma para 17 idiomas, utilizando técnicas como CountVectorizer para la extracción de características ([Language Detection using NLP](#)).

## 3. Modelos de Aprendizaje Profundo:

- Los métodos más avanzados emplean modelos de aprendizaje profundo para detectar idiomas, incluso en textos que mezclan varios idiomas. [FastText](#), desarrollado por Facebook AI Research, es un ejemplo sobresaliente. Este modelo utiliza redes neuronales para la clasificación de texto y es altamente eficiente, capaz de identificar hasta 176 idiomas. Está disponible como una biblioteca de código abierto y se utiliza en aplicaciones prácticas de detección de idioma ([FastText](#)).

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 ejempl

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

# 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=False)
print(result)

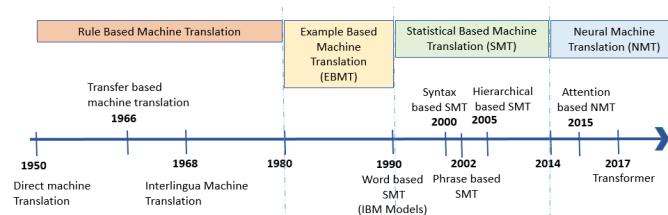
# Imprime:
# {'lang': 'de', 'score': 0.9996129870414734}
# {'lang': 'fr', 'score': 0.992135226726532}
```

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](#).

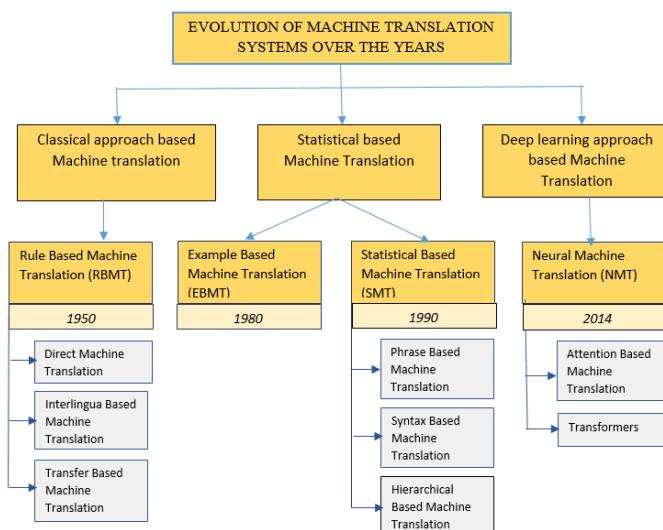
## 8. Traducción de texto (Language Translation)

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:



Fuente: <https://www.mdpi.com/2079-9292/12/7/1716>

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 t  
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  
print(translated_word)  
  
translated_word = PonsTranslator(source='english', target='spanish').translate  
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').transla  
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.

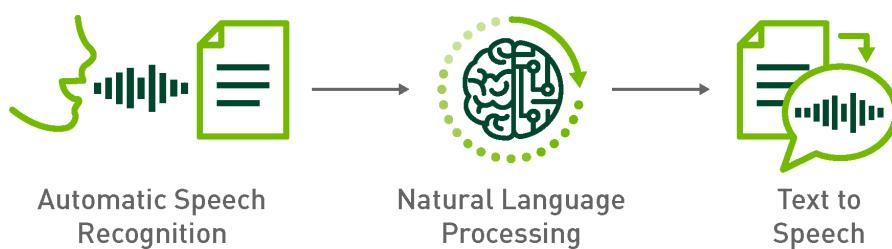
```
from deep_translator import MyMemoryTranslator  
translated = MyMemoryTranslator(source='en', target='zh').translate(text='cu  
print(translated)
```

Otra librería que soporta múltiples traductores es [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'))
```

## 9. 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.

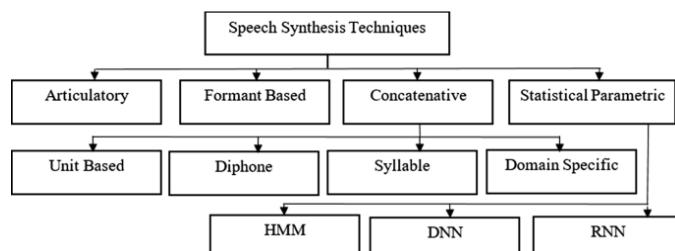
### 2. Generación de Sonido:

- 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

## 10. 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.

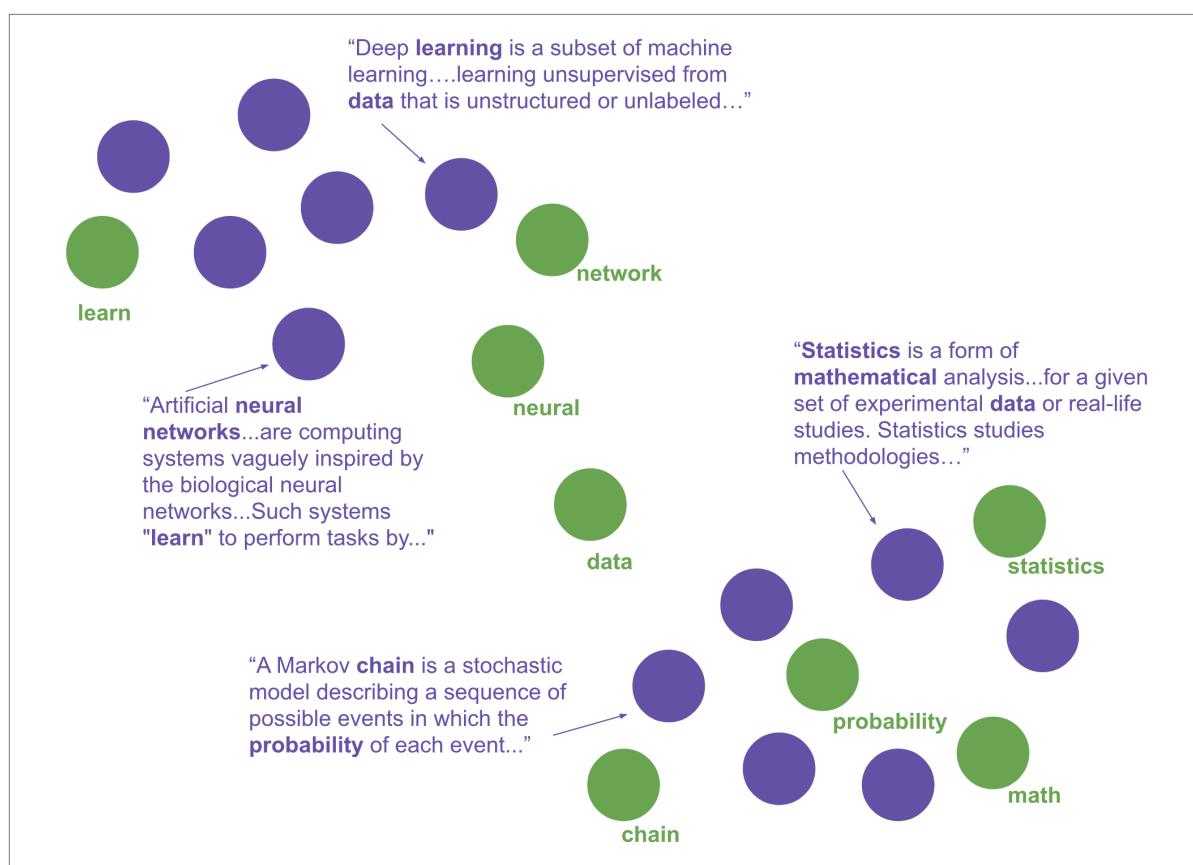
Algunas técnicas comunes de modelado de tópicos incluyen Latent Dirichlet Allocation (LDA), Non-negative Matrix Factorization (NMF), y recientemente modelos basados en deep learning como LDA2Vec (<https://github.com/cemoody/lda2vec>) y Top2Vec (<https://github.com/ddangelov/Top2Vec>).

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

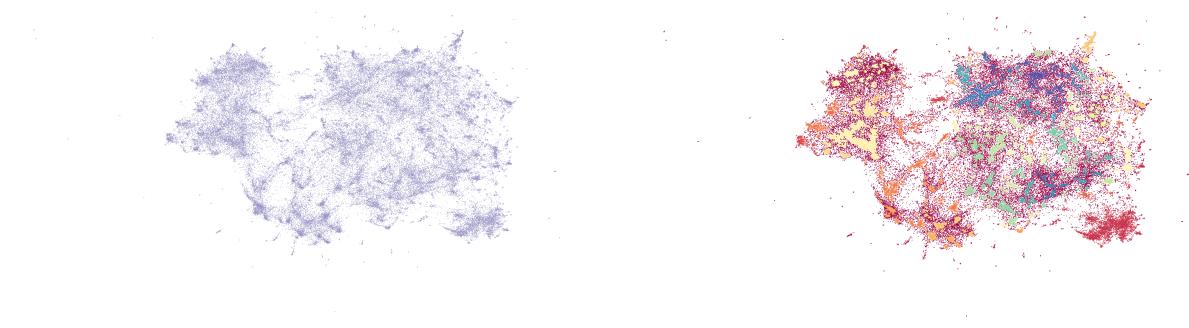
Después de tener vectores para cada documento, el siguiente paso natural sería dividirlos en grupos utilizando un algoritmo de agrupación. Sin embargo, los vectores generados en el primer paso pueden tener hasta 512

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

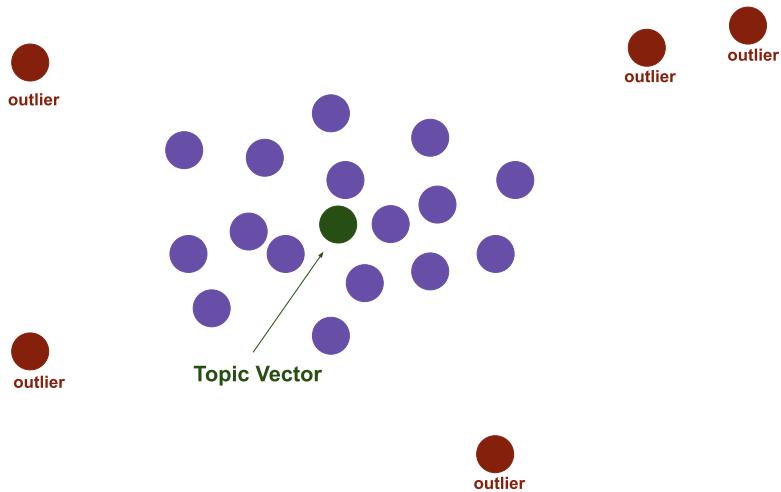
[HDBSCAN](#), un algoritmo de agrupación (clustering) basado en densidad jerárquica, para encontrar áreas densas de documentos. HDBSCAN es básicamente una extensión del algoritmo [DBSCAN](#) que lo convierte en un algoritmo de agrupación jerárquica. Utilizar HDBSCAN para modelado de temas tiene sentido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



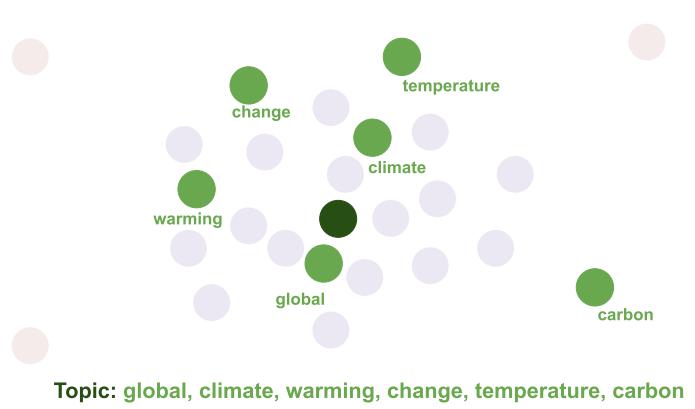
Se crean embeddings de los documentos de una menor dimensión, usando UMAP. Los vectores de documentos en un espacio de alta dimensión son muy dispersos; la reducción de dimensión ayuda a encontrar áreas densas. Cada punto es un vector de documento.

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 violeta 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_transformers]`

Utilizaremos 3 datasets de noticias, obtenidos de la página:

<https://huggingface.co/datasets?sort=trending&search=noticias>

```
#!pip install torch sentence_transformers top2vec[sentence_transformers] dat  
  
from datasets import load_dataset  
import numpy as np  
import pandas as pd  
from top2vec import Top2Vec  
  
# Descarga de datasets de noticias  
dataset_1 = load_dataset("BrauuHdzM/noticias-en-espanol")  
dataset_2 = load_dataset("Nicky0007/titulos_noticias_rcn_clasificadas")  
dataset_3 = load_dataset("Nicky0007/cointelegraph_noticias_Es")  
  
# Obtener la columna Contenido y acumular en una lista  
contents = dataset_1['train']['Contenido'] + dataset_2['train']['text'] + dataset_3['train']['text']  
  
# Crear un modelo Top2Vec usando el modelo de embedding "paraphrase-mi  
hdbSCAN_args = {'min_cluster_size': 100,  
                 'min_samples': 5,  
                 'metric': 'euclidean',  
                 'cluster_selection_method': 'eom'}  
model = Top2Vec(documents=contents, workers=8, embedding_model="para
```

En el ejemplo, creamos una variables `contents` 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 `paraphrase`

`multilingual-MiniLM-L12-v2`, 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.

Los parámetros `hdbscan_args` se utilizan para configurar el algoritmo HDBSCAN (Clustering Espacial Basado en Densidad Jerárquica de Aplicaciones con Ruido), que es utilizado por Top2Vec para el agrupamiento de datos. Vamos a desglosar cada parámetro:

1. `min_cluster_size` : 100

- Establece el tamaño mínimo de los clusters.
- Significa que cada cluster debe contener al menos 100 puntos de datos (documentos en este caso).
- Valores más altos resultarán en menos clusters pero más grandes.

2. `min_samples` : 5

- Determina el número de muestras en una vecindad para que un punto sea considerado un punto núcleo.
- Está relacionado con la densidad requerida para que una región sea considerada densa.
- Valores más altos hacen que el algoritmo sea más conservador al encontrar puntos núcleo.

3. `metric` : 'euclidean'

- Especifica la métrica de distancia a utilizar para medir la distancia entre puntos.
- 'euclidean' se refiere a la distancia euclíadiana, que es la distancia en línea recta "ordinaria" entre dos puntos en un espacio euclidiano.

4. `cluster_selection_method` : 'eom'

- Determina cómo se extraen los clusters de la jerarquía de clusters.
- 'eom' significa "Exceso de Masa", que es generalmente el método preferido.
- Tiende a producir un número menor de clusters más significativos semánticamente en comparación con el método alternativo 'leaf'.

Estos parámetros le permiten al modelo, ajustar el proceso de clustering.

Luego utilizaremos el modelo Top2Vec para buscar tópicos relacionados con las palabras clave "messi", "futbolista", "argentino". 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`:

```
# Mostrar la cantidad de documentos
print("Documentos:", len(contents))

# Mostrar la cantidad total de tópicos
print("Cantidad de tópicos:", model.get_num_topics())

# Si queremos obtener todos los tópicos y sus palabras:
topic_words, word_scores, topic_nums = model.get_topics()

# Buscar tópicos relacionados con las palabras claves "messi", "futbolista", "argentino"
# topic_words: Para cada tópico se devuelven las 50 palabras principales, en orden de similitud.
# word_scores: Para cada tópico se devuelven los puntajes de similitud coseno.
# topic_scores: Para cada tópico se devolverá la similitud coseno con las palabras clave.
# topic_nums: Se devolverá el índice único de cada tópico.
topic_words, word_scores, topic_scores, topic_nums = model.search_topics(k=3)

# Mostrar las palabras clave, las puntuaciones de los tópicos y los 2 documentos más relevantes para cada tópico
for i, topic_num in enumerate(topic_nums):
    print(f"\nTópico {i + 1}:")
    print(f"Palabras clave: {', '.join(topic_words[i])}")
    print(f"Puntuación del tópico: {topic_scores[i]}")

# Obtener los 2 documentos más relevantes para cada tópico
documents, document_scores, document_ids = model.search_documents_k_nearest(topic_nums, k=2)

# Imprimir los documentos más relevantes
for doc_id, document, score in zip(document_ids, documents, document_scores):
    print(f"\nDocumento ID {doc_id} - Score: {score}:")
```

```
print(document[:500]) # Mostrar una previsualización de los primeros 50 documentos
print("\n" + "-"*80)
```

Y obtendremos lo siguiente:

Documentos: 28740

Cantidad de tópicos: 53

Tópico 1:

Palabras clave: futbol, brasileño, fifa, brasil, futbolista, futbolistas, argentinos,

Puntuación del tópico: 0.6174974236456289

Documento ID 14791 - Score: 0.7843004465103149:

Catar 2022 La leyenda del fútbol que se separó de su pareja por... ¡ver un partid

Documento ID 16889 - Score: 0.7826096415519714:

Catar 2022 ¿Se acabó el amor? Prensa costarricense exige la salida de Luis F

---

Tópico 2:

Palabras clave: cuba, artistas, alberto, espanol, publica, existen, historia, histo

Puntuación del tópico: 0.4370446388940895

Documento ID 4042 - Score: 0.7418169975280762:

La pintura está de regreso. Por lo menos, eso se percibe después de visitar la

Documento ID 352 - Score: 0.7391732335090637:

Los sueños, aspiraciones y deseos del pintor mexicano Nicolás Cuéllar (1927-

---

Tópico 3:

Palabras clave: venezuela, colombianos, colombiana, colombiano, colombia, la

Puntuación del tópico: 0.3426288379346375

Documento ID 9760 - Score: 0.838007926940918:

Los inmigrantes venezolanos lograron lo que no había podido la alta diplomac

Documento ID 2855 - Score: 0.8233909010887146:

Los presidentes de Colombia, Gustavo Petro, y Venezuela, Nicolás Maduro, pl

Una vez que tenemos los tópicos, podemos graficar nubes con las relevancias de las palabras clave:

```
print(model.topic_words[0])  
model.generate_topic_wordcloud(0)
```

## Topic 0

