



Facultad de Ciencias Exactas, Ingeniería y

Agrimensura (FCEIA - UNR)

Trabajo Práctico 1

IA4.2 Procesamiento del Lenguaje Natural

Tecnicatura Universitaria en Inteligencia Artificial

Demarré Lucas
Donnarumma Cesar
Fontana Gustavo Julián

03/11/2023

Ejercicio 1:

Para el primer ejercicio la consigna pide construir un dataset utilizando técnicas de web scraping de páginas web de nuestra elección, definiendo 4 categorías de noticias/artículos y extrayendo por categoría 10 textos con los siguientes datos: url, título y texto. Con esto hay que construir un dataset en formato csv.

Lo que se hizo en primer lugar fue plantear las categorías que para nuestro caso fueron textos de reseñas/críticas de videojuegos, películas, música y hoteles.

En segundo lugar por cada categoría se construyó un diccionario (con claves numéricas del 0 al 11) cuyos primeros 10 elementos corresponden a urls de donde se extraerán los textos y los siguientes dos elementos corresponden al nombre de la clase de la etiqueta html que contiene los datos y el nombre de la categoría en sí.

Cada uno de los diccionarios se guarda en una lista.

```
hoteles = {
  0: 'https://www.unmundopequenio.com/resena-la-merced-del-alto-cachi/',
  1: 'https://www.unmundopequenio.com/resena-hotel-apraxis-san-petersburgo/',
  2: 'https://www.unmundopequenio.com/resena-gia-vien-hotel-ho-chi-minh-city/',
  3: 'https://www.unmundopequenio.com/havana-hotel-el-cairo/',
  4: 'https://www.unmundopequenio.com/resena-balcon-de-la-plaza-salta/',
  5: 'https://www.unmundopequenio.com/resena-tara-place-bangkok/',
  6: 'https://www.unmundopequenio.com/resena-jj-bungalows-phi-phi/',
  7: 'https://www.unmundopequenio.com/resena-travelodge-fort-myers/',
  8: 'https://www.unmundopequenio.com/resena-comfort-inn-long-island-city-nueva-york/',
  9: 'https://www.unmundopequenio.com/albergo-enrica-hotel-en-roma/',
  10: 'entry-content',
  11: 'hoteles'
}

urls = [videojuegos, peliculas, musica, hoteles]
```

El siguiente paso fue crear una función (llamada web_scraping) que, a partir de una url y la clase de la etiqueta que contiene los datos (parámetros), descarga los datos de dicha página web y retorna el título y el texto.

```
def web_scrapping(url, clase):

    ''' Descarga, extrae y retorna de las paginas web de la lista "urls" los titulos y textos de los respectivos articulos'''

    response = requests.get(url, verify=True) #LO PASÉ A TRUE PARA QUE VERIFIQUE CREDENCIALES Y NO MUESTRE LA ADVERTENCIA EN EL OTRO BLOQUE

    soup = BeautifulSoup(response.text, 'html.parser')

    titulo = soup.find('h1')

    texto = soup.find('div', {'class': clase})

    texto = texto.find_all(['p', 'h2'])

    parrafo = '\n'.join(par.text for par in texto)

    return titulo.text, parrafo
```

Previos a ser exportados como csv los datos se guardaron en un DataFrame de Pandas, para lo cual creamos uno vacío con las columnas: url, título, texto y categoría.

```
# Lista de columnas (vacías) para crear luego un df
data = {'url': [], 'titulo': [], 'texto': [], 'categoria': []}

# DataFrame con columnas vacío
dataset = pd.DataFrame(data)
```

Y luego de esto comienza la parte de descargar de manera sistemática los textos y guardarlos en nuestro DF.

Se recorre la lista que contiene los diccionarios de urls de cada categoría, y a su vez se recorren los diccionarios llamando por cada url a la función web_scrapping (que extrae los datos y devuelve título y texto) para posteriormente guardarlos como una nueva fila del DF.

```
# Iteramos sobre lista de urls
for categoria in urls:
    # Tomamos cada elemento de cada categoria, a excepcion de los dos ultimos que contienen: la clase de la etiqueta que contiene el cuerpo del articulo
    # y el nombre de la categoria
    for i in range( len(categoria) - 2 ):

        # Extraemos el titulo y el articulo en si
        titulo, texto = web_scrapping(categoria[i], categoria[10])
        # Extraemos el nombre de la categoria
        nombre_categoria = categoria[11]

        # Creamos un diccionario con los valores de la nueva fila
        nueva_fila = {'url': categoria[i], 'titulo': titulo, 'texto': texto, 'categoria': nombre_categoria}

        # Lo agregamos al df
        dataset = dataset.append(nueva_fila, ignore_index=True)
```

Una vez que ya se terminó con cada categoría solo queda exportar el DF Pandas a csv.

```
# Exportamos el df como .csv
dataset.to_csv('articulos.csv', index=False)
```

Ejercicio 2:

El segundo ejercicio pedía que a partir de los datos descargados en el punto anterior se entrene un modelo de clasificación de noticias en categorías específicas.

En primer lugar se procedió a hacer una limpieza de los datos previo a la vectorización codificando la variable independiente (categorías), eliminando los signos de puntuación, convirtiendo los títulos a minúscula, eliminando los acentos y también eliminando algunas palabras como crítica y reseña ya que estaban bastante presente en muchos títulos de algunas categorías y se consideró que podrían influir demasiado en el resultado final a la hora de clasificar sin que necesariamente sean palabras verdaderamente representativas de la categoría. Sobre el final de todo este proceso se separó los datos en variable dependiente (X) y variable independiente (Y).

```
# 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')

# Copiamos lo que nos interesa del dataset
datos = dataset[['titulo', 'categoria']].copy()

# Codificación de variable independiente
labels = { "videojuegos": 0, "películas": 1, "musica": 2,
          "hoteles": 3}

# Mapeo de categoria a cuantitativa
datos['categoria'] = datos['categoria'].map(labels)

# Eliminamos signos de puntuación
datos['titulo'] = datos['titulo'].str.replace('[^\w\s]', '')

# Convertimos a minúsculas
datos['titulo'] = datos['titulo'].str.lower()

# Eliminamos acentos para homogeneizar el texto
import unicodedata

def remove_accents(input_str):
    nfkd_form = unicodedata.normalize('NFKD', input_str)
    return ''.join([c for c in nfkd_form if not unicodedata.combining(c)])

datos['titulo'] = datos['titulo'].apply(lambda titulo: remove_accents(titulo))

# Eliminamos algunas palabras que pueden ser comunes a todos los títulos a raíz de ver en un primer intento que tenían demasiada influencia,
# cuando en realidad son palabras que no tienen nada que ver con ninguna de las categorías mas alla del tipo de texto del que se trata
datos['titulo'] = datos['titulo'].str.replace('reseña', '').str.replace('crítica', '')

X = datos['titulo']
y = datos['categoria']
```

El siguiente paso fue vectorizar, para este primer ejemplo se utilizó TF-IDF eliminando las palabras de parada, dividir en train y test y entrenar un modelo de regresión logística.

```
# Vectorización de los textos con eliminación de palabras vacías
vectorizer = TfidfVectorizer(stop_words=spanish_stop_words)
X_vectorized = vectorizer.fit_transform(X)

# División del dataset
X_train, X_test, y_train, y_test = train_test_split(X_vectorized, y, test_size=0.2, random_state=45)

# Creación y entrenamiento del modelo de Regresión Logística
modelo_LR = LogisticRegression(max_iter=1000)
modelo_LR.fit(X_train, y_train)
```

Observando las métricas de entrenamiento podemos ver que el modelo se ajustó muy bien a los datos de entrenamiento.

```
Precisión Regresión Logística: 1.0
Reporte de clasificación Regresión Logística:
      precision    recall  f1-score   support

0         1.00      1.00      1.00         6
1         1.00      1.00      1.00         8
2         1.00      1.00      1.00         9
3         1.00      1.00      1.00         9

 accuracy         1.00
macro avg         1.00
weighted avg         1.00
```

Pero observando las de prueba se puede ver un desempeño muy malo del modelo.

```
Precisión Regresión Logística: 0.25
Reporte de clasificación Regresión Logística:
      precision    recall  f1-score   support

0         1.00      0.00      0.00         4
1         0.00      0.00      0.00         2
2         0.25      1.00      0.40         1
3         0.33      1.00      0.50         1

 accuracy         0.25
macro avg         0.23
weighted avg         0.11
```

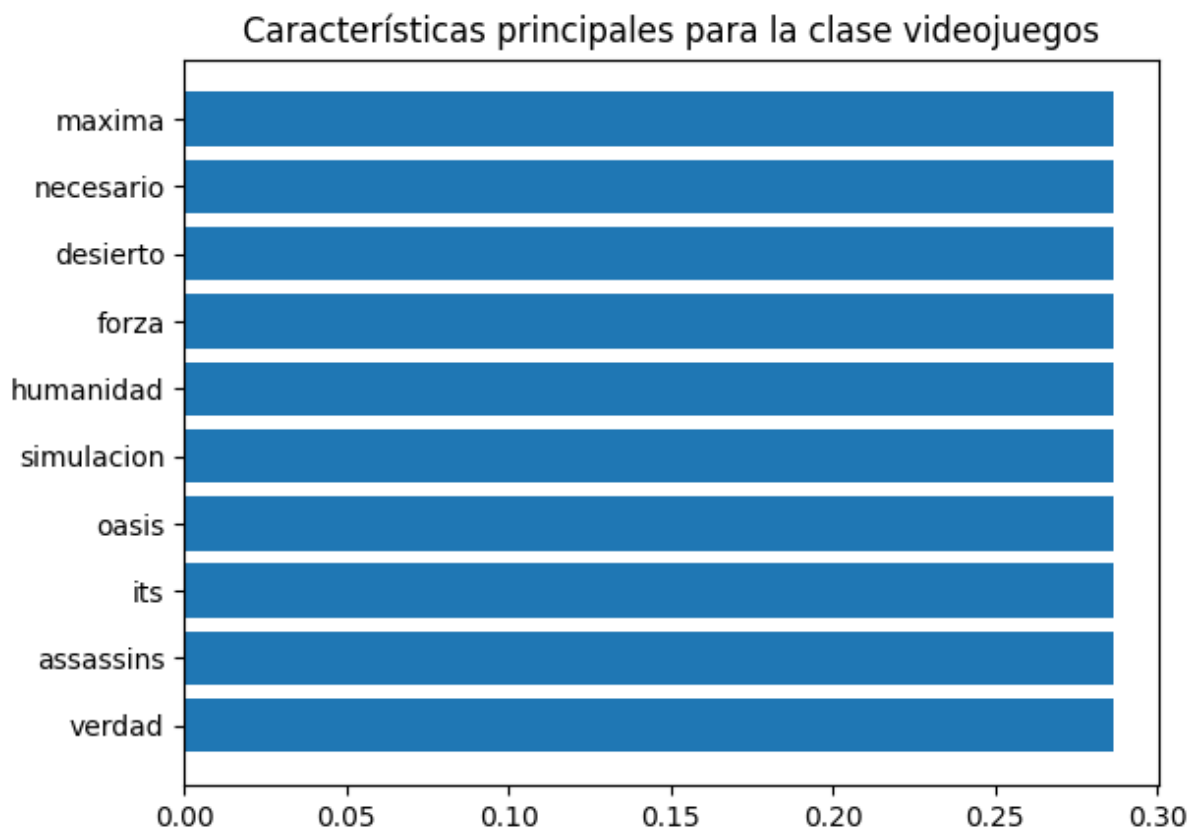
En general las métricas de test están muy por debajo de las métricas de train lo que podría indicar sobreajuste.

Esto se verifica mejor a la hora de hacer predicciones donde el modelo tiende a clasificar muy mal. No logró clasificar bien ningún título de videojuegos ni de películas, clasificó bien los de música y tiende a clasificar mucho como hoteles.

```
La frase 'Reseña Killer Frequency: Desata el terror de los años 80' pertenece a la categoría: musica
La frase 'Reseña LEGO 2K Drive, construye, corre y disfruta' pertenece a la categoría: hoteles
La frase 'Reseña Cannon Dancer: Osman, un poco de nostalgia' pertenece a la categoría: hoteles
La frase 'Reseña: Hotel Audran (París)' pertenece a la categoría: hoteles
La frase 'Reseña: Nefertiti Hotel (Luxor)' pertenece a la categoría: hoteles
La frase 'Crítica de 'Divertimento', una 'feel-good movie' francesa basada en hechos reales' pertenece a la categoría: hoteles
La frase 'Crítica de El poder del perro' pertenece a la categoría: hoteles
La frase 'Crítica de Parásitos' pertenece a la categoría: hoteles
La frase 'Banda: MORS SUBITA Disco: Origin of Fire Sello: Out of Line Music Año: 2023' pertenece a la categoría: musica
La frase 'Grupo: MERCENARY. Disco: "Soundtrack for the End Times". Sello: NoiseArt Records. Año: 2023.' pertenece a la categoría: musica
La frase 'Banda: RED CAIN Disco: NÆ BLISS Sello: Autoeditado Año: 2023' pertenece a la categoría: musica
```

Claramente estamos ante a una situación donde estamos entrenando un modelo con muy pocos datos como para que pueda generalizar bien cada categoría, son muy pocos ejemplos en general. Por ejemplo luego de dividir en train y test para la categoría 0 entrenamos solo con 6 filas. Además nuestro dataset si bien presenta distintas categorías todas pertenecen al género reseñas lo que podría también hacer más difícil aún la clasificación.

Esto se verifica aún más cuando vemos las palabras más influyentes en el modelo para la categoría videojuegos por ejemplo. Seguramente en una situación de gran cantidad de datos de entrenamiento palabras como máxima, necesario, desierto, no estarían como las primeras.



Sobre el final con TF-IDF nos dimos cuenta que a la hora de clasificar nuevos títulos hay un problema de pérdida de datos ya que al entrenar nuestro vectorizador con los datos iniciales de entrenamiento y test se genera un diccionario de dimensión $x = \text{cantidad palabras en el corpus}$ y se crean los vectores correspondientes en este espacio. Cuando queremos hacer una nueva clasificación y vectorizar nuevos títulos utilizando el mismo vectorizador estamos en la situación de que hay palabras que no están presentes en nuestro diccionario por lo que estaríamos perdiendo información.

Entonces se decidió probar vectorizar con BERT para ver si se obtenían mejores resultados. Básicamente se realizó previamente la misma normalización y limpieza de datos y se entrenó un modelo de regresión logística nuevamente.

Esta vez en test y train se obtuvieron las mismas métricas, todo con 1 lo cual también llama la atención, aparentemente el modelo sería perfecto lo cual no es muy creíble.

```
Precisión Regresión Logística: 1.0
Reporte de clasificación Regresión Logística:
      precision    recall  f1-score   support

     0         1.00      1.00      1.00         1
     1         1.00      1.00      1.00         4
     2         1.00      1.00      1.00         2
     3         1.00      1.00      1.00         1

 accuracy         1.00
 macro avg         1.00
 weighted avg         1.00
```

Y finalmente se realizan las mismas predicciones que antes. Esta vez se obtienen mejores resultados en categorías como videojuegos que antes no aparecían, mismos resultados en música pero el modelo sigue tendiendo a clasificar como hoteles cosas que no son. La categoría de películas no pudo clasificar en ningún momento.

```
Texto: 'Reseña Killer Frequency: Desata el terror de los años 80'  
Clasificación predicha: videojuegos  
  
Texto: 'Reseña LEGO 2K Drive, construye, corre y disfruta'  
Clasificación predicha: videojuegos  
  
Texto: 'Reseña Cannon Dancer: Osman, un poco de nostalgia'  
Clasificación predicha: videojuegos  
  
Texto: 'Reseña: Hotel Audran (París)'  
Clasificación predicha: hoteles  
  
Texto: 'Reseña: Nefertiti Hotel (Luxor)'  
Clasificación predicha: hoteles  
  
Texto: 'Crítica de 'Divertimento', una 'feel-good movie' francesa basada en hechos reales'  
Clasificación predicha: películas  
  
Texto: 'Crítica de El poder del perro'  
Clasificación predicha: hoteles  
  
Texto: 'Crítica de Parásitos'  
Clasificación predicha: hoteles  
  
Texto: 'Banda: MORS SUBITA Disco: Origin of Fire Sello: Out of Line Music Año: 2023'  
Clasificación predicha: musica  
  
Texto: 'Grupo: MERCENARY. Disco: "Soundtrack for the End Times". Sello: NoiseArt Records. Año: 2023.'  
Clasificación predicha: musica  
  
Texto: 'Banda: RED CAIN Disco: NÆE'BLISS Sello: Autoeditado Año: 2023'  
Clasificación predicha: musica
```

Conclusiones: por una cuestión de resultados en las predicciones, métricas y por la cuestión de la pérdida de datos a la hora de vectorizar se prefiere BERT sobre TF-IDF, aunque con ninguna se obtuvieron muy buenos resultados en general. Hay que tener en cuenta también que el modelo fue hecho dentro de todo rápidamente como un ejercicio más dentro de un conjunto y no se le pudo dedicar el tiempo necesario como para intentar lograr un buen clasificador, además de que hay pocos datos de entrenamiento y las métricas obtenidas no son demasiado robustas, se podrían haber utilizado técnicas como validación cruzada para una mayor fiabilidad.

Ejercicio 3:

En el ejercicio 3 se solicita procesar el texto limpiándolo y normalizándolo, y luego mostrar la importancia de las palabras a través de una nube de palabras.

En primer lugar se crea un diccionario 'textos_por_categoria' que va a almacenar un par, clave:valor, donde las categorías únicas del dataset van a ser las claves, y una lista con los textos de cada categoría puntal sus valores. Luego este diccionario va a ser utilizado por una función en un paso posterior.

```
# Crear un diccionario para almacenar los textos por categoría
textos_por_categoria = {}

# Iterar a través de las categorías y extraer los textos correspondientes
for categoria in archivo.categoria.unique():
    # Filtrar el DataFrame para la categoría actual
    df_categoria_actual = archivo[archivo['categoria'] == categoria]

    # Extraer los textos de la categoría actual y los almacena en una lista
    textos_categoria_actual = df_categoria_actual['texto'].tolist()

    # Almacena la lista de textos en el diccionario usando la categoría como clave
    textos_por_categoria[categoria] = textos_categoria_actual
```

En segundo lugar se declara una función 'remove_stopwords' que permite eliminar las palabras de parada. Aquellas que no tienen mucha relevancia en un texto lo que nos permite centrarnos en aquellas que tienen una mayor importancia. Esta función recibe como argumento un texto.

Para la eliminación de stopwords se utiliza la librería NLTK, siendo necesario descargar el conjunto de stopwords en español, con el cual luego se compara cada palabra del texto y se filtran aquellas que no pertenezcan. Finalmente se obtiene el conjunto de palabras relevantes para el texto.

```
# Función para eliminar stopwords de una frase
def remove_stopwords(text):
    word_tokens = word_tokenize(text)
    filtered_text = [word for word in word_tokens if word.casefold() not in stop_words]
    return " ".join(filtered_text)
```

En tercer lugar se genera la función 'textos_por_categoria' que va finalmente a limpiar el texto. Recibe como argumento los textos de una categoría determinada.

En esta función se realizan varias tareas:

- Con la librería RE y el uso de expresiones regulares se eliminan los links, caracteres especiales, números y enlaces de fotos.
- Con la librería demoji se eliminan los emoticonos.
- Se llama a la función 'remove_stopwords' descrita previamente para eliminar stopwords.
- Con la librería spaCy se lematiza el texto reduciendo las palabras a su raíz en el idioma trabajado.

Además dentro de la función se obtienen y almacena la frecuencia de las palabras ya procesadas, y el total de palabras finales.

Finalmente la función retorna el total de palabras, la frecuencia, y el texto limpio y normalizado.


```

#Función para procesar el texto
def procesar_texto(textos_por_categoria):
    #Se carga la función para lematizar
    nlp = es_core_news_sm.load()

    # Creamos un objeto FreqDist para las palabras
    fdist_words = FreqDist()

    for texto in textos_por_categoria:
        #Elimino enlaces web
        texto = re.sub(r'http\S+|www\S+|\.com\b', '', texto)

        #Se quitan los emojis del texto
        sin_emojis = demoji.replace(texto, '')

        #Se quitan caracteres especiales
        sin_caracteres_especiales = re.sub(r'^\w\s+|\d+', '', sin_emojis)

        #Se lematiza para obtener la raíz de cada palabra
        texto_lematizado = [token.lemma_.lower() for token in nlp(sin_caracteres_especiales)]
        texto_lematizado = ' '.join(texto_lematizado)

        #Se eliminan las stopwords
        sin_stop_words = remove_stopwords(texto_lematizado)

        # Eliminar menciones a "pictwitter"
        sin_stop_words = re.sub(r'pictwitter\S*', '', sin_stop_words)

        #Se cuenta la frecuencia de las palabras
        words = word_tokenize(sin_stop_words)

        # Actualizo FreqDist
        fdist_words.update(words)

    # Se guardan el total de palabras y las frecuencias
    total_palabras = f'Total de palabras: {len(fdist_words)}'
    frecuencia = dict(fdist_words)
    return total_palabras, dict(sorted(frecuencia.items())), sin_stop_words

```

Por último se presenta una función para generar nubes de palabras. La función recibe como argumento las categorías y luego itera por cada una de ellas y genera un subplot con el texto normalizado de cada categoría obtenido como resultado de invocar a la función 'textos_por_categoria'.

```

#Función para generar nubes de palabras
def generar_nube_palabras(categorias):
    n = 0
    plt.figure(figsize = (14,14), facecolor = None)
    for categoria in categorias:
        plt.subplot(141+n)
        wordcloud = WordCloud(width = 800, height = 800,
                               background_color = 'white',
                               stopwords = None,
                               min_font_size = 10).generate(procesar_texto(textos_por_categoria[categoria])[2])

        plt.axis("off")
        plt.title(f'Categoría {categoria}', color='navy', fontweight='bold', fontsize=12, pad=15)
        plt.tight_layout(pad = 1.5)
        plt.imshow(wordcloud)
        n += 1
    plt.show()

```

Conclusiones: Luego de procesar los datos eliminando texto innecesario, caracteres especiales, emojis, etc, se obtuvo el texto que nos interesa. A través de la nube de palabras generada por la función de ploteo, se pueden visualizar aquellas palabras más

representativas y con más frecuencia en base a su tamaño en el gráfico. Llamando a la función que nos procesa el texto obtenemos el total de palabras y la frecuencia para cada una de ellas luego de aplicar las transformaciones. Como es de esperar ciertas palabras relacionadas a la categoría que pertenecen tienen una mayor frecuencia. Es necesario comentar que la representatividad de ciertas palabras no se vuelve tan evidente en un conjunto tan pequeño de datos.

Ejemplo de salida final del procesamiento de texto:

| procesar_texto(textos_por_categoria['peliculas'][:2]) | procesar_texto(textos_por_categoria['videojuegos'][:2]) |
|--|--|
| <pre>(Total de palabras: 2128', {'abandonado': 1, 'abandonar': 2, 'abonado': 1, 'abordar': 1, 'abrazar': 1, 'abrir': 2, 'absolutamente': 2, 'absurdo': 1, 'abuela': 1, 'abuelo': 1, 'abusar': 1, 'acabar': 5, 'academia': 1, 'accec': 2, 'accesible': 1,</pre> | <pre>(Total de palabras: 2197', {'abajo': 7, 'abasi': 1, 'abierto': 1, 'abolladura': 1, 'abordar': 1, 'abr': 1, 'abrazar': 1, 'abrebocas': 1, 'abrir': 2, 'abróchate': 1, 'abuelo': 3, 'acabar': 2, 'acariciaste': 1, 'acceder': 3, 'acceso': 2,</pre> |

Salida de nube de palabras:



Ejercicio 4: Introducción

En este ejercicio, hemos optado por analizar la similitud entre los títulos de noticias pertenecientes a la categoría **Videojuegos**. Con este propósito, empleamos cuatro modelos distintos. Esto nos permite observar cómo los diferentes enfoques pueden generar resultados diversos al abordar el mismo problema. Luego de aplicar cada modelo, examinamos la similitud entre los títulos utilizando un mapa de calor, ya que consideramos que esta es la representación visual más adecuada para este tipo de análisis ya que en dicho mapa, los colores indican el grado de similitud entre las oraciones, y las etiquetas de los ejes X e Y se asignan a los títulos de las oraciones en la lista original. Los colores más oscuros representan una mayor similitud. A continuación, vamos a explicar cada paso.

Variables

Primero vienen las variables que vamos a utilizar para cada modelo. *chosen_category* es la primera variable, esta representa la categoría elegida para el ejercicio. *data_ej4* representa el Dataset filtrado del original con solo las noticias de la categoría elegida. *titles* la filtración de los títulos de cada noticia.

```
# Separamos los títulos de una categoría que elijamos
chosen_category = 'videojuegos'
data_ej4 = dataset[dataset['categoria'] == chosen_category]
titles = data_ej4['titulo'] # Solo almacenamos los títulos de dicha
categoría
```

Luego creamos una lista, *titles_list*, para guardar los títulos del Dataset. Para ellos empleamos de un bucle en donde, en cada iteración, le quitamos los caracteres especiales, así como las tildes y convertimos en título a minúscula.

```
titles_list = []
for title in titles:
    title = re.sub(r'[!,¿;\'?]', '', title.lower()) # Eliminamos signos de
exclamación, interrogación, etc. y convertimos a minúscula
    title = unidecode(title) # Eliminamos las tildes
    titles_list.append(title) # Guardamos dentro de una lista los títulos
para facilitar el manejo de los mismos
```

Ahora pasemos a explicar cada modelo.

Modelos

Promedio de Word Embeddings: Este modelo tiene, lo que podría describirse como, una aproximación más ingenua al problema.

Para comenzar, cargamos un modelo de procesamiento de lenguaje natural proporcionado por SpaCy. Luego, creamos una lista denominada *similarities* con el propósito de conservar las similitudes entre los títulos. A través de dos bucles anidados, comparamos cada par de títulos en la lista y calculamos su nivel de similitud utilizando el modelo SpaCy. Estos valores de similitud son posteriormente registrados en una matriz bidimensional.

```
# Cargamos el modelo de procesamiento de lenguaje natural
nlp = spacy.load("en_core_web_md")

# Creamos una lista para almacenar las similitudes
similarities = []

# Iteramos a través de la lista de títulos
for i in titles_list:
    row = []

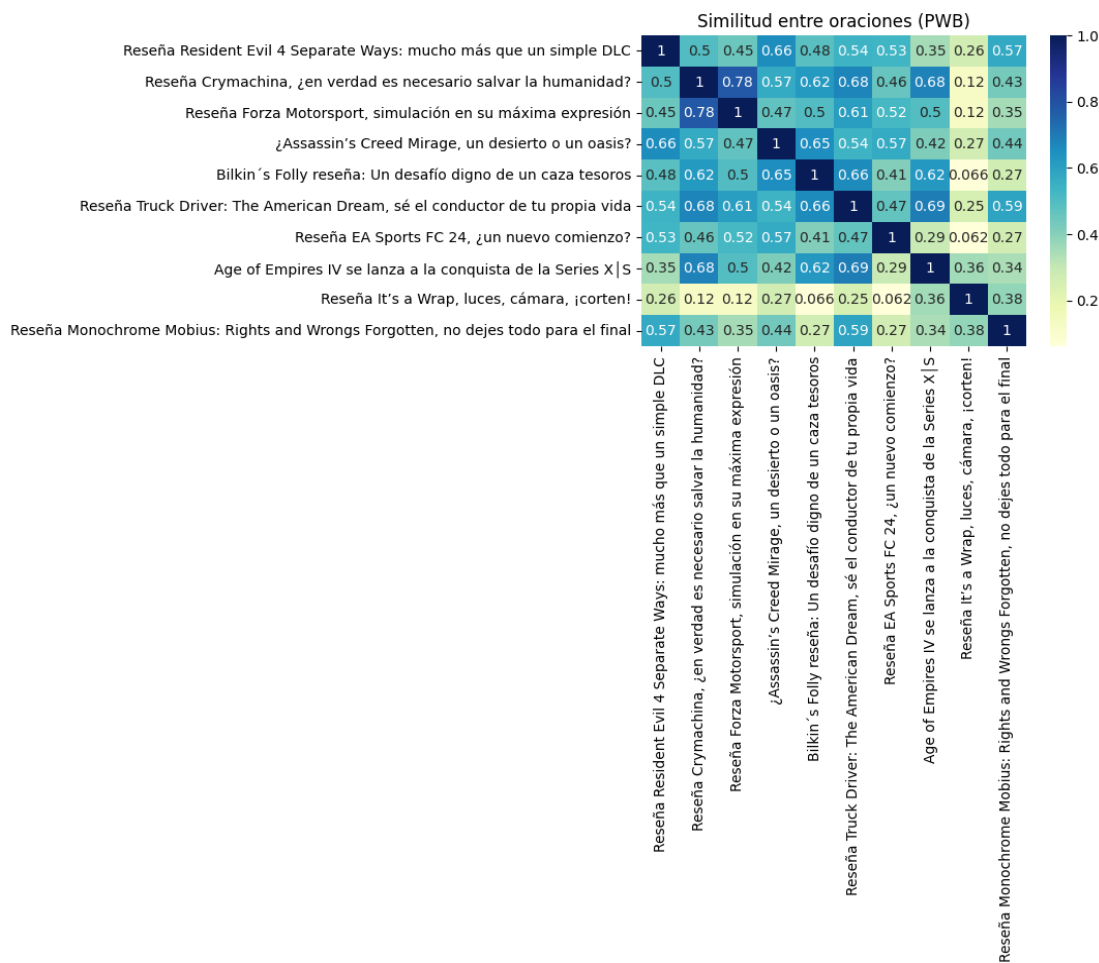
    # Iteramos nuevamente a través de la lista de títulos para comparar con
    cada título
    for x in titles_list:
        # Calculamos la similitud entre los títulos utilizando el modelo
        SpaCy
        sim = nlp(i).similarity(nlp(x))
```

```
# Agregamos el valor de similitud a la fila
row.append(sim)
```

```
# Agregamos la fila de similitudes a la lista de similitudes
similarities.append(row)
```

Finalmente, generamos el mapa de calor a partir de esta matriz, en el cual los colores reflejan el grado de similitud entre los títulos.

```
# Creamos un mapa de calor de la matriz de similitud
plt.figure(figsize=(6, 4))
sns.heatmap(similarities, annot=True, xticklabels=titles, yticklabels=titles,
cmap="YlGnBu")
plt.title("Similitud entre oraciones (PWB)")
plt.show()
```



Universal Sentence Encoder (USE): En esta parte realizamos una serie de operaciones para visualizar la similitud entre los títulos utilizando *embeddings* generadas por el modelo **Universal Sentence Encoder (USE)** multilingüe.

Comenzamos definiendo una función llamada *visualize_similarity*. Esta función toma dos conjuntos de embeddings, dos listas de etiquetas (los títulos de noticias) y un título para el gráfico.

Definición de la función para visualizar la similitud entre embeddings y etiquetas

```
def visualize_similarity(embeddings_1, embeddings_2, labels_1, labels_2,
                        plot_title,
                        plot_width=1200, plot_height=600,
                        xaxis_font_size='12pt', yaxis_font_size='12pt'):
```

Primero, se verifica que la longitud de los conjuntos de embeddings y las etiquetas sea la misma para garantizar la correspondencia adecuada entre ellos.

```
# Verificación de igual longitud de embeddings y etiquetas
assert len(embeddings_1) == len(labels_1)
assert len(embeddings_2) == len(labels_2)
```

Luego, se calcula la similitud entre los embeddings utilizando la similitud coseno y se ajusta la escala de similitud para que esté en el rango de 0 a 1.

```
# Cálculo de similitud de texto basado en similitud coseno
sim = 1 - np.arccos(
    sklearn.metrics.pairwise.cosine_similarity(embeddings_1,
                                                embeddings_2))/np.pi
```

```
# Preparación de datos para el gráfico
embeddings_1_col, embeddings_2_col, sim_col = [], [], []
for i in range(len(embeddings_1)):
    for j in range(len(embeddings_2)):
        embeddings_1_col.append(labels_1[i])
        embeddings_2_col.append(labels_2[j])
        sim_col.append(sim[i][j])
df = pd.DataFrame(zip(embeddings_1_col, embeddings_2_col, sim_col),
                  columns=['embeddings_1', 'embeddings_2', 'sim'])
```

Luego, se preparan los datos para crear el mapa de calor. Se crean listas que almacenan las etiquetas de los embeddings y los valores de similitud entre todos los pares de títulos. Configuramos un mapeador de colores para asignar colores a las celdas del mapa de calor y luego crear el gráfico.

```
# Configuración del mapeo de colores
mapper = bokeh.models.LinearColorMapper(
    palette=[*reversed(bokeh.palettes.YlOrRd[9])], low=df.sim.min(),
    high=df.sim.max())

# Creación del gráfico interactivo
p = bokeh.plotting.figure(title=plot_title, x_range=labels_1,
                          x_axis_location="above",
                          y_range=[*reversed(labels_2)],
                          width=plot_width, height=plot_height,
                          tools="save", toolbar_location='below', tooltips=[
                              ('pair', '@embeddings_1 ||| @embeddings_2'),
                              ('sim', '@sim')])
p.rect(x="embeddings_1", y="embeddings_2", width=1, height=1, source=df,
       fill_color={'field': 'sim', 'transform': mapper}, line_color=None)

# Configuración de aspectos visuales del gráfico
p.title.text_font_size = '12pt'
p.axis.axis_line_color = None
p.axis.major_tick_line_color = None
p.axis.major_label_standoff = 16
p.xaxis.major_label_text_font_size = xaxis_font_size
p.xaxis.major_label_orientation = 0.25 * np.pi
```

```
p.yaxis.major_label_text_font_size = yaxis_font_size
p.min_border_right = 300

# Mostramos el gráfico
bokeh.io.output_notebook()
bokeh.io.show(p)
```

Después de definir la función *visualize_similarity*, el código carga un modelo **USE multilingüe** desde una URL específica. Luego, utilizamos este modelo para obtener embeddings de los títulos en una lista llamada *titles_list*.

```
# URL del modelo 'USE' multilingüe
module_url =
'https://tfhub.dev/google/universal-sentence-encoder-multilingual/3'

# Cargamos el modelo 'USE' multilingüe
model = hub.load(module_url)

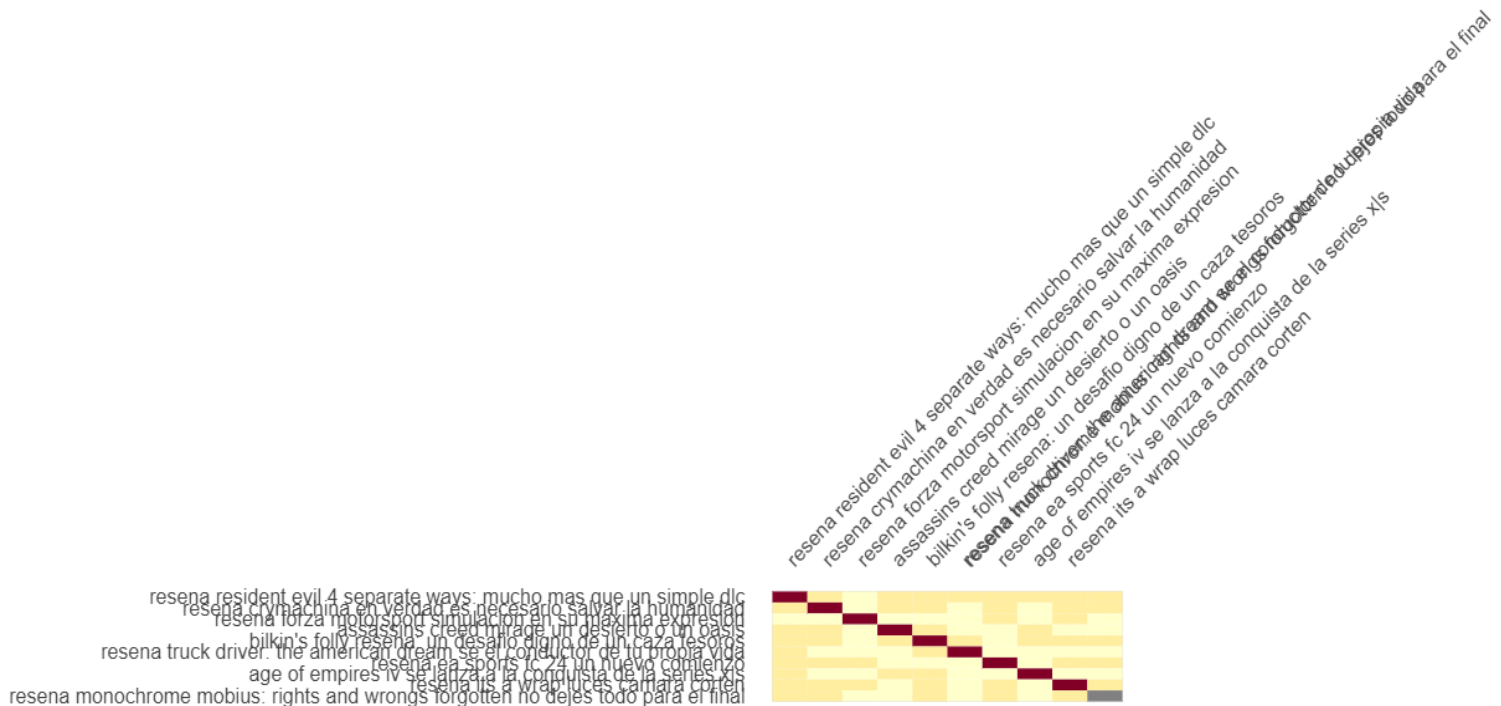
# Función para obtener embeddings de texto
def embed_text(input):
    return model(input)

# Obtención de embeddings de los títulos en la lista
embed_titles = embed_text(titles_list)
```

Finalmente, llamamos a la función *visualize_similarity* para visualizar la similitud entre los títulos utilizando el modelo **USE** y generamos el mapa de calor interactivo.

```
# Llamamos a la función para visualizar la similitud entre los títulos
utilizando embeddings
visualize_similarity(embed_titles, embed_titles, titles_list, titles_list,
'Similitud entre títulos (USE)')
```

Similitud entre títulos (USE)



Doc2Vec: En esta parte llevamos a cabo un proceso para evaluar la similitud los títulos utilizando el modelo **Doc2Vec**.

Comenzamos por tokenizar y etiquetar los datos. Cada título se tokeniza en palabras y se le asigna una etiqueta única, lo que crea una lista de objetos *TaggedDocument* que contiene las palabras tokenizadas y las etiquetas.

```
# Tokenizamos los datos y los etiquetamos
tagged_data = [TaggedDocument(words=word_tokenize(_d.lower()), tags=[str(i)])
for i, _d in enumerate(titles_list)]
```

Luego, se configuran los parámetros para el modelo **Doc2Vec**, incluyendo el tamaño del vector de características *vector_size*, el tamaño de la ventana *window*, el recuento mínimo de palabras *min_count*, el número de trabajadores para el procesamiento paralelo *workers*, y el número de épocas de entrenamiento *epochs*.

```
# Configuramos los parámetros para el modelo
model = Doc2Vec(vector_size=100, window=5, min_count=1, workers=4,
epochs=1000)
```

A continuación, se construye el vocabulario del modelo mediante el método *build_vocab*, que toma los datos etiquetados y prepara el modelo para el entrenamiento. El modelo se entrena utilizando los datos etiquetados mediante el método *train*, y se realizan un total de **1000** épocas de entrenamiento.

```
# Construimos el vocabulario
model.build_vocab(tagged_data)
```



```
# Entrenamos el modelo
model.train(tagged_data, total_examples=model.corpus_count,
epochs=model.epochs)
```

Después de entrenar el modelo, se calcula la similitud entre los títulos. Para cada título en la lista, se obtiene un vector de características utilizando *infer_vector*. Luego, se utilizan los vectores de características para calcular las similitudes con todos los demás títulos, y estos valores se almacenan en la lista *similarities_d2v*. Se ordena la lista *similarities_d2v* para que cada sublista esté ordenada por el índice de las tuplas, lo que representa la similitud de cada título con los demás.

```
similarities_d2v = []
for i in titles_list:
    vector = model.infer_vector(word_tokenize(str(i).lower()))

    similares = model.dv.most_similar(vector) # Vemos las similitudes de cada
título con todos los demás
    similarities_d2v.append(similares) # Los guardamos en una lista

# Ordenamos cada sublista según el índice de cada tupla
sorted_similarities = [sorted(sublist, key=lambda x: int(x[0])) for sublist
in similarities_d2v]
```

A continuación, se extrae el segundo elemento de cada tupla en las sublistas, que representa la similitud de cada título con los demás. Esto crea una lista llamada *heatmap_similarities*. Finalmente, se crea un mapa de calor.

```
# Luego separamos de cada tupla el segundo elemento de cada sublista, este
representa la similitud de cada título con los demás
heatmap_similarities = [[tup[1] for tup in sublist] for sublist in
sorted_similarities]
```

```
# Creamos un mapa de calor
plt.figure(figsize=(6, 4))
sns.heatmap(heatmap_similarities, annot=True, xticklabels=titles_list,
yticklabels=titles_list, cmap="YlGnBu")
plt.title("Similitud entre oraciones (Doc2Vec)")
plt.show()
```



Sentence-BERT (S-BERT): En esta última parte evaluamos la similitud semántica entre un conjunto de oraciones utilizando un modelo preentrenado de **Sentence-BERT (S-BERT)** llamado **all-mpnet-base-v2**.

Primero, se aplica el modelo para codificar las oraciones que se encuentran en la lista *titles_list*. Cada oración se convierte en un vector de características que representa su contenido de manera numérica.

```
# Cargamos el modelo preentrenado all-mpnet-base-v2
modelo = SentenceTransformer('all-mpnet-base-v2')

# Codificamos las oraciones
embeddings = modelo.encode(titles_list, convert_to_tensor=True)
```

Posteriormente, se calculan las puntuaciones de similitud entre las oraciones utilizando la métrica de similitud coseno. Esto da como resultado una matriz de similitud donde cada valor representa cuán similares son dos oraciones en términos de significado.

```
# Calculamos las puntuaciones de similitud
cosine_scores = util.cos_sim(embeddings, embeddings)
```

Finalmente, para visualizar estas similitudes, generamos el mapa de calor como en los modelos anteriores.

```
# Creamos un mapa de calor
plt.figure(figsize=(6, 4))
sns.heatmap(cosine_scores, annot=True, xticklabels=titles_list,
yticklabels=titles_list, cmap="YlGnBu")
plt.title("Similitud entre oraciones (S-BERT)")
plt.show()
```



Conclusión: Cada modelo emplea diversas técnicas para analizar la semántica de cada título y determinar si son similares o no. Todos estos modelos varían en el enfoque que se le da al problema, y, en consecuencia, producen resultados diferentes. Sin embargo, al examinar todos los títulos en la categoría de videojuegos, notamos que hay numerosos títulos que siguen la estructura de 'Reseñas a...'. En otras palabras, muchos títulos comparten un tema común: la reseña de un videojuego, algunos con comentarios adicionales en el título y otros sin estos.

Para nosotros, es evidente que estos títulos son muy similares, ya que todos se centran en la misma temática y tratan sobre el acto de reseñar un videojuego siendo el comentario adicional irrelevante para la similitud en sí. Sin embargo, los modelos no logran captar esta similitud, ya que carecen de la capacidad de comprender el contexto detrás de cada título sin acceder al contenido de la noticia, a diferencia de nosotros.

Ejercicio 5:

En el ejercicio 5 se solicita crear un programa interactivo que reciba una categoría seleccionada por un usuario y devuelva el resumen de las noticias incluidas en ella.

Para cumplir el objetivo se procesa el texto para obtener un resumen coherente y conciso. Se lleva a cabo un enfoque extractivo con el fin de conservar la mayor parte del texto original ya que se tratan de reseñas.

En primer lugar, para realizar el resumen extractivo se utiliza la función 'summarize' que utiliza la librería networkx para crear un grafo a partir de una matriz de similaridad (hecha con una librería llamada 'cosine_similarity' que devuelve una matriz con la relación de dos palabras de acuerdo al ángulo entre sus vectores). Luego se aplica PageRank al grafo y se almacena el resultado en un diccionario con las frases como claves y sus pesos (score de PageRank) como valores ordenados de menor a mayor. Finalmente se retornan las frases que tienen mejor puntaje.

```
[ ] # Función para generar un resumen extractivo usando PageRank
def summarize(similarity_matrix, original_sentences, num_sentences=5):
    # Crear un grafo a partir de la matriz de similitud
    nx_graph = nx.from_numpy_array(similarity_matrix)
    # Aplicar PageRank al grafo
    scores = nx.pagerank(nx_graph, alpha=0.1) #MODIFIQUE EL ALFA PORQUE NO CONVERGÍA
    # Ordenar las oraciones por su puntuación y seleccionar las mejores
    ranked_sentences = sorted(((scores[i], s) for i, s in enumerate(original_sentences)), reverse=True)
    return ' '.join([ranked_sentences[i][1] for i in range(num_sentences)])
```

En segundo lugar, se crea una función 'resumir' que recibe a una categoría como argumento y que utiliza al modelo pre-entrenado de la librería spaCy para obtener los embeddings de las palabras y frases, lematizar el texto y quitar las stopwords. Además se define la matriz de similaridad con las oraciones filtradas y se pasa como argumento a la función 'summarize' que se encarga de realizar el resumen extractivo mencionado previamente. Este resumen fue almacenado durante el ciclo implementado en un diccionario que guarda a la reseña particular como valor y al título correspondiente como clave. Finalmente se retorna la

visualización del resumen para cada una de las noticias de la categoría en un formato agradable.

```
def resumir(categoria):
    nlp = spacy.load('es_core_news_md')
    texto_por_categoria = archivo[archivo['categoria'] == categoria]
    texto_por_categoria.reset_index(inplace=True)

    # Lemmatizar y eliminar stopwords de cada oración
    resumen_global = {}
    nro_noticia = 0
    for texto in texto_por_categoria.texto:
        lemmatized_sentences = []
        original_sentences = []
        doc = nlp(texto)
        for sent in doc.sents:
            lemmatized_sentence = " ".join([token.lemma_ for token in sent if not token.is_stop and not token.is_punct])
            if lemmatized_sentence.strip() != '': # Asegurarse de que la oración no esté vacía
                lemmatized_sentences.append(lemmatized_sentence)
                original_sentences.append(str(sent).strip())

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

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

        # Crear una matriz de similitud entre las oraciones filtradas
        similarity_matrix = cosine_similarity(sentence_vectors)
        resumen = summarize(similarity_matrix, original_sentences, num_sentences=2)
        resumen_global[texto_por_categoria.titulo[nro_noticia]] = resumen
        nro_noticia += 1

    for key, values in resumen_global.items():
        html_text = f'<div style="width:100%">{values}</div>'
        print(f'{key}:')
        display(HTML(html_text))
        print()
```

Por último se diseñó un programa interactivo para el usuario que despliega un menú y permite seleccionar cada una de las categorías disponibles del conjunto de datos y obtener el resumen de las noticias disponibles para cada categoría.

Formato del menú:

```
> dar_resumen()

Ingrese el número de la opción seleccionada:
[1] Películas
[2] Musica
[3] Hoteles
[4] Videojuegos

Ingrese cualquier otro valor para salir


```

Ejemplo de salida del resumen:

```
-----Resumen de la categoría películas-----
Crítica de 'Mi otro Jon', comedia solidaria con Carmen Maura y Aitana Sánchez-Gijón:
Esta afirmación parece contradictoria al referirse a un cineasta que rueda películas familiares, y que se adelantó en más de un lustro, con 'Maktub' (2011), su ópera prima, al auge actual del que goza este. Apuesta, como en el resto de sus cintas, por un humor blanco y muestra de nuevo su predilección por la ciencia ficción y el poder de lo sobrenatural.

Crítica de 'Killers of the Flower Moon': un monumental Scorsese da a Leonardo DiCaprio el papel de su vida:
Su mirada descarnada al Reino de Terror que vivió la Nación Osage, formada por nativos americanos, en la década de 1920, apenas tiene parangón en la historia del cine estadounidense, o en la propia obra de Martin Scorsese. La magnífica 'Killers of the Flower Moon', la nueva película de Martin Scorsese, podría leerse como la prueba de un posible cambio de paradigma, en el que el Monte Rushmore del cine yanqui, históricamente ocupado por un cine oficialista y edificante, pasaría a ser conquistado por las películas bastardas.
```

Opcional: Bot de Telegram. [Link del bot](#)

ADVERTENCIA: Se necesita la función *summarize* del **Ejercicio 5** para funcionar.

Introducción

En este último ejercicio decidimos hacer un Bot de Telegram el cual entrega un resumen de noticias del blog de *elpais.com*. En dicho Bot, al mandar el comando `/news` el Bot imprime una lista de títulos pertenecientes a cada noticia para que el usuario escoja y, posteriormente, mandar el resumen de dicha noticia. A continuación, procedemos a explicar las funciones necesarias para que el Bot funcione.

Funciones

La primera función es, *df_create*, tiene como objetivo recopilar el contenido de noticias a partir de una lista de URLs de noticias. La función realiza varios pasos para lograr esto. Primero, se crea un diccionario llamado *news_content* para almacenar el contenido de cada noticia, donde las claves son los títulos de las noticias y los valores son los contenidos respectivos.

```
def df_create(news):
    # Un diccionario para almacenar el contenido de cada noticia
    news_content = {}
```

Luego, la función recorre la lista de URLs proporcionada como entrada y utiliza la biblioteca *requests* para hacer solicitudes HTTP a cada URL. Si la respuesta es satisfactoria (código de estado 200), se procede a analizar la página web utilizando *BeautifulSoup* para extraer información.

```
# Recorremos la lista de URLs y extraemos el contenido
for key, url in news.items():
    response = requests.get(url)
    if response.status_code == 200:
        soup = BeautifulSoup(response.text, 'html.parser')
```

Se buscan elementos HTML que contengan tanto el título de la noticia como su contenido. Si se encuentran tanto el título como el contenido, se seleccionan todos los elementos `<p>` (párrafos) dentro del cuerpo de la noticia, se extrae el texto de cada párrafo y se concatenan para formar el texto completo del contenido de la noticia.

```

        # Encontramos los elementos que contienen el contenido de la
noticia
        title = soup.find('h1', class_='a_t')
        content = soup.find('div', attrs={'data-dtm-region':
'articulo_cuerpo'})

        if content and title:
            # Seleccionamos todos los elementos <p> dentro del cuerpo de
la noticia
            paragraphs = content.find_all('p')

```

La función agrega este contenido al diccionario *news_content*, utilizando el título de la noticia como clave y el contenido como valor. Si no se puede encontrar el contenido, se agrega un mensaje indicando que no se pudo encontrar el contenido de la noticia.

```

        # Texto completo del contenido de la noticia
        news_article_content = ''.join(paragraph.text for paragraph
in paragraphs)
        news_content[title.text] = news_article_content
    else:
        news_content[title.text] = "No se pudo encontrar el contenido
de esta noticia."

```

Finalmente, la función crea un DataFrame a partir de los datos recopilados en el diccionario *news_content*. El DataFrame consta de dos columnas: "title" (título de la noticia) y "content" (contenido de la noticia). El resultado es un DataFrame que contiene el título y el contenido de cada noticia, lo que facilitará su posterior manejo.

```

return pd.DataFrame(list(news_content.items()), columns=['title',
'content'])

```

La siguiente función es prácticamente igual a la función *resumir* que se utiliza en el **Ejercicio 5**, por lo que su explicación se encontrará allí. La única diferencia notable es que, en lugar de producir un resumen para todo un DataFrame de noticias, esta función genera un resumen para una noticia individual contenida en un DataFrame.

```

def summarize_news(title, df):
    nlp = spacy.load('es_core_news_md')
    contenido = df[df['title'] == title]['content'].values[0]

    # Lematizar y eliminar stopwords de cada oración
    lemmatized_sentences = []
    original_sentences = []
    doc = nlp(contenido)

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

```

```

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

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

# Crear una matriz de similitud entre las oraciones filtradas
similarity_matrix = cosine_similarity(sentence_vectors)
summary = summarize(similarity_matrix, original_sentences,
num_sentences=2)

return summary

```

Variables

Luego viene la creación de las variables necesarias para la ejecución del programa. Por un lado, tenemos un diccionario con 20 noticias provenientes del blog en cuestión.

```

# Diccionario con las url de las noticias
news = {
    0: 'https://elpais.com/babelia/2023-09-23/la-escritura-como-exoesqueleto.html',
    1: 'https://elpais.com/educacion/escuelas-en-red/2023-09-17/una-educacion-grande-en-un-pueblo-pequeno.html',
    2: 'https://elpais.com/cultura/del-tirador-a-la-ciudad/2023-08-01/existe-un-mundo-mejor-pero-es-mas-carro.html',
    3: 'https://elpais.com/cultura/del-tirador-a-la-ciudad/2023-07-11/hacer-hueco-a-los-estudiantes-para-cambiar-la-ciudad.html',
    4: 'https://elpais.com/cultura/del-tirador-a-la-ciudad/2023-07-05/sonar-con-una-piscina.html',
    5: 'https://elpais.com/mamas-papas/expertos/2023-07-04/carmen-osorio-periodista-es-dificil-que-a-los-ninos-no-les-afecten-las-opiniones-de-otros-y-las-vidas-idilicas-que-ven-en-internet.html',
    6: 'https://elpais.com/elviajero/viajero-astuto/2023-06-30/viajes-de-una-vida.html',
    7: 'https://elpais.com/elviajero/2023-06-26/guia-para-viajar-a-singapur-con-ninos-consejos-para-llegar-y-atracciones-imprescindibles.html',
    8: 'https://elpais.com/educacion/escuelas-en-red/2023-06-25/mirando-al-mundo-con-curiosidad.html',
    9: 'https://elpais.com/educacion/2023-06-11/el-instituto-de-torrejocillo-abre-su-pinacoteca-al-publico.html',
    10: 'https://elpais.com/opinion/2023-06-07/no-se-venden-bajos.html',
    11: 'https://elpais.com/television/2023-06-05/pescar-datos-en-un-mar-de-documentacion.html',
    12: 'https://elpais.com/elviajero/2023-05-20/consejos-de-un-trotamundos-para-solucionar-los-imprevistos-que-surgen-al-viajar.html',

```

```

13:
'https://elpais.com/elviajero/viajes-paco-nadal/2023-03-28/tienes-un-movil-de-
-miles-de-megapixeles-pero-sabes-hacer-buenas-fotos-de-tus-viajes.html',
14:
'https://elpais.com/elviajero/2023-03-25/claves-para-hacer-la-maleta-de-los-n
-inos-para-irse-de-viaje.html',
15:
'https://elpais.com/elviajero/viajero-astuto/2023-03-21/todo-el-mundo-cabe-en
-un-huerto-de-la-region-del-loira.html',
16:
'https://elpais.com/planeta-futuro/escuelas-en-red/2023-03-19/profesores-que-
apuestan-por-mejorar-la-calidad-de-la-ensenanza.html',
17:
'https://elpais.com/elviajero/gastronotas-de-capel/2023-03-04/de-verdad-exist
-en-los-espaguetis-a-la-bolonesa.html',
18:
'https://elpais.com/elviajero/viajes-paco-nadal/2023-03-01/skrei-pescando-en-
-el-artico-noruego-el-bacalao-mas-viajero.html',
19:
'https://elpais.com/elviajero/gastronotas-de-capel/2023-02-26/llega-el-moment
o-de-las-galeras-un-marisco-tan-feo-como-sabroso-estrena-temporada.html',
}

```

Y por el otro, la creación del DataFrame utilizando la función `df_create` explicada con anterioridad y el diccionario de noticias establecido.

```

# Dataframe de las noticias. Primera columna: 'title', segunda columna:
'content'
news_dataframe = df_create(news)

```

Implementación del bot

Por último, pasemos a explicar cómo se implementa y ejecuta el Bot de Telegram que brindara a los usuarios resúmenes de las noticias que elijan dentro de un listado ya predeterminado.

Comenzamos estableciendo un token de acceso del Bot y recopilando una lista de títulos de noticias del DataFrame `news_dataframe`. Luego, utilizamos la biblioteca `telebot` para inicializar el Bot

```

# Token de acceso del bot
bot_token = '6351031143:AAGW5Oyax3ubR1bxwn-Xug3ELUyJ7VnaSmU'

# Guardamos todos los títulos en una lista
titles = news_dataframe['title'].values.tolist()

# Inicializamos el bot
bot = telebot.TeleBot(bot_token)

```

El Bot responde a varios comandos y acciones del usuario. Cuando un usuario envía los comandos `/start` o `/help`, el bot le da la bienvenida y proporciona instrucciones sobre cómo obtener resúmenes de noticias utilizando el comando `/news`. Al enviar `/news`, el Bot responde con un mensaje que enumera los títulos de las noticias disponibles.

```

# Mensaje de bienvenida
@bot.message_handler(commands=['start', 'help'])

```

```

def send_welcome(message):
    bot.reply_to(message, ";Bienvenido a tu bot de noticias! Envía /news para recibir un resumen de noticias.")

# Mandamos la lista de noticias
@bot.message_handler(commands=['news'])
def send_news_summary(message):
    # Creamos un mensaje con los títulos de las noticias
    user_message = "Elige una noticia:\n\n"
    for i, title in enumerate(titles, 1):
        user_message += f"{i}. {title}\n"

    # Enviamos el mensaje al usuario
    bot.send_message(message.chat.id, user_message)

```

El manejo principal ocurre cuando un usuario elige un número de noticia de la lista. El bot verifica si la elección es válida y, si es así, obtiene el título de la noticia seleccionada para generar el resumen con la función `summarize_news`. El resumen se envía al usuario como respuesta.

```

# Manejamos la elección del usuario
@bot.message_handler(func=lambda message: True)
def handle_user_choice(message):
    try:
        # Convertimos el mensaje del usuario en un número
        election = int(message.text)

        # Verificamos si la elección es válida
        if 1 <= election <= len(titles):
            # Obtenemos el título de la noticia seleccionada
            selected_title = titles[election - 1]

            # Con el título seleccionado obtenemos el resumen de esa noticia
            # con la función 'summarize_news'
            summary = summarize_news(selected_title, news_dataframe)
            bot.send_message(message.chat.id, f"Resumen de
            '{selected_title}':\n{summary}")
        else:
            bot.send_message(message.chat.id, "Selección no válida. Por favor, ingresa un número válido.")
        except ValueError:
            bot.send_message(message.chat.id, "Por favor, ingresa un número válido para elegir una noticia.")

```

Finalmente, la función `bot.polling()` mantiene el bot encendido y funcionando siempre y cuando el código se esté ejecutando, de otra forma, el bot no funcionará.

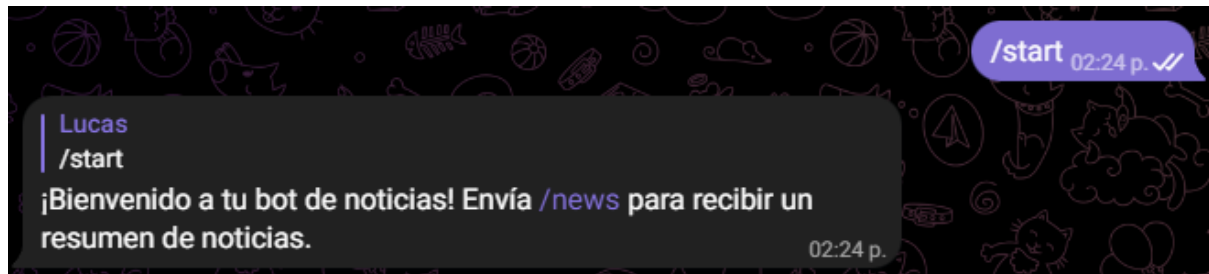
```

# Ejecuta el bot
bot.polling()

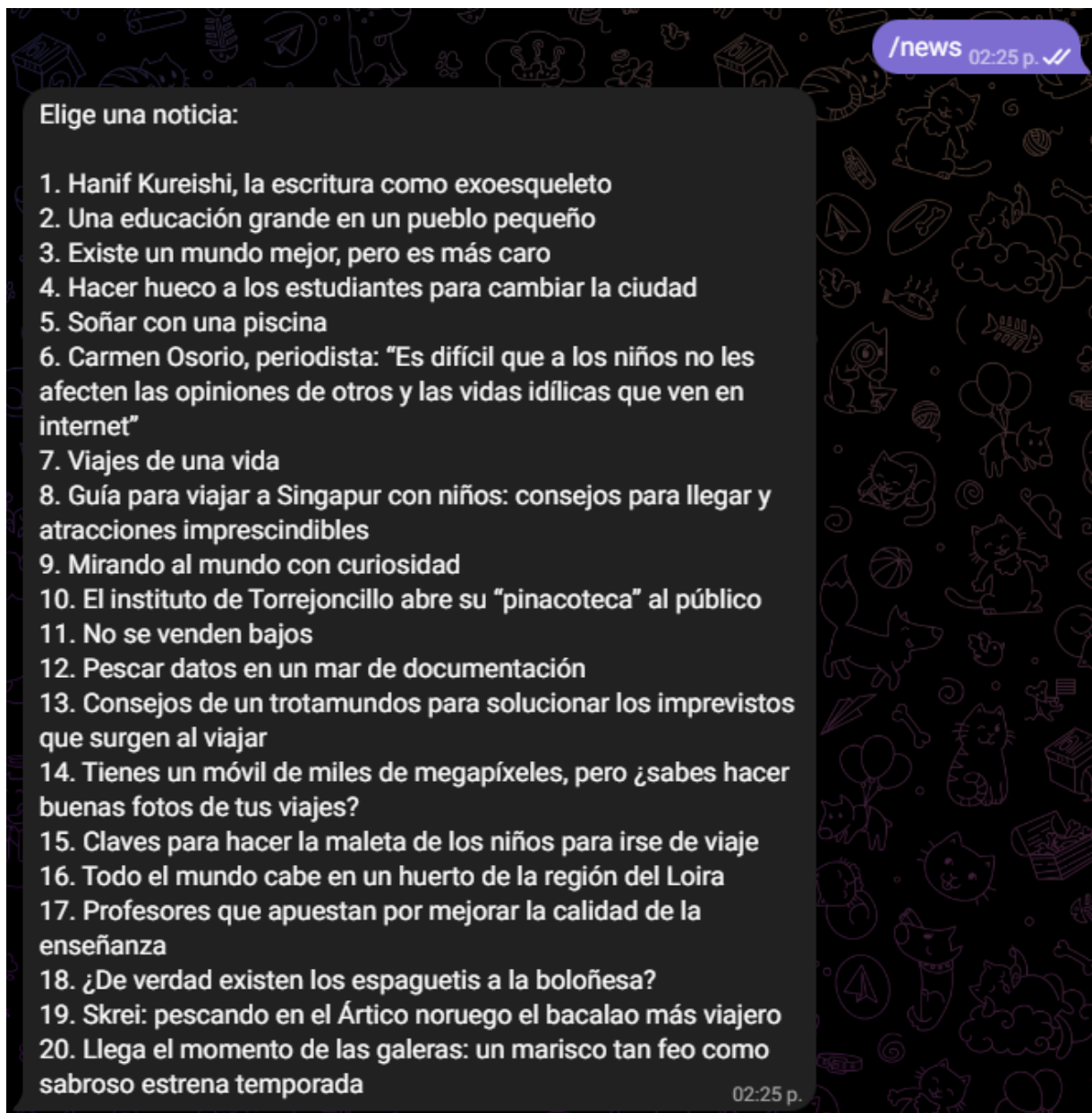
```


Ejemplos de comandos

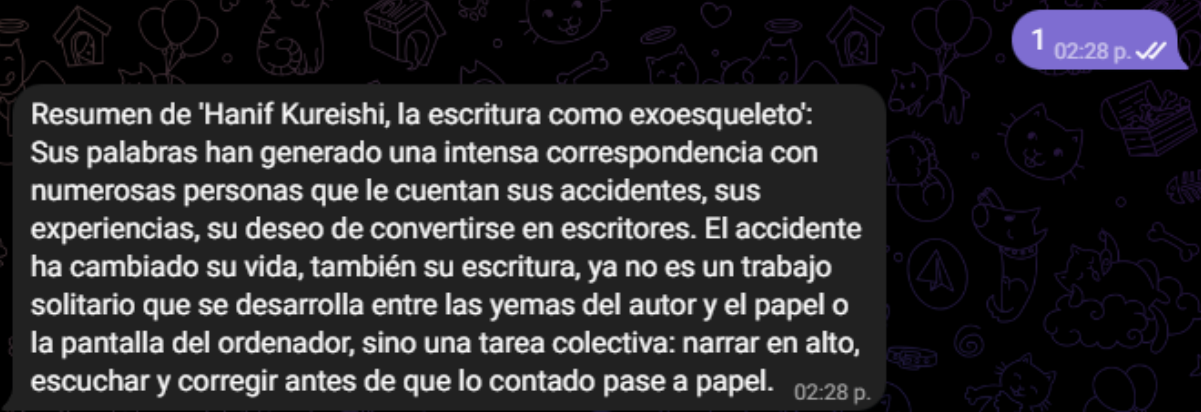
/start



/news



Elegir opción/Resumen



1 02:28 p. ✓

Resumen de 'Hanif Kureishi, la escritura como exoesqueleto':
Sus palabras han generado una intensa correspondencia con numerosas personas que le cuentan sus accidentes, sus experiencias, su deseo de convertirse en escritores. El accidente ha cambiado su vida, también su escritura, ya no es un trabajo solitario que se desarrolla entre las yemas del autor y el papel o la pantalla del ordenador, sino una tarea colectiva: narrar en alto, escuchar y corregir antes de que lo contado pase a papel.

02:28 p.