



Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Tecnicatura Universitaria en Inteligencia Artificial
Procesamiento del lenguaje natural

TRABAJO PRÁCTICO N°1 - Año 2024 - 2° Semestre

Alumnos:

Antuña, Franco A-4637/1

Gallardo, Jonatan G-5970/6

Profesores:

Juan Pablo Manson

Alan Geary

Resumen.....	3
Introducción.....	3
Objetivos específicos.....	3
Fuentes de datos.....	4
Desarrollo.....	5
Implementación.....	6
Preprocesado.....	6
Clase TextPreprocessor.....	6
Carga.....	8
Función load_books.....	8
Función load_words.....	11
Función load_datasets_union.....	12
Función count_words.....	14
Función procesar_entrada_usuario:.....	15
Modelos.....	15
Función model_sentiments.....	15
Función train_model.....	18
Función model_recommender.....	19
Predicciones.....	20
Función recommend_item.....	20
Función predict_sentiments.....	21
Función predict_recomendation.....	22
Bot:.....	23
Función input_user.....	23
Función print_slow.....	25
Función main.....	25
Función chatbot.....	28
Ejemplos de uso:.....	30
Conclusiones:.....	31
Puntos de mejora:.....	32
Referencias.....	33

Resumen

En este informe se detalla el funcionamiento y el propósito de cada función de nuestro algoritmo. El sistema ha sido estructurado en varias funciones con el objetivo de ofrecer una recomendación personalizada entre tres actividades: leer un libro, ver una película o jugar a un juego de mesa. La elección final es adaptada a la preferencia de actividad, sugiriendo un título específico de acuerdo con la disposición emocional del usuario.

Para desarrollar estas recomendaciones, se emplean varios conjuntos de datos. Se utiliza un conjunto de datos que contiene información sobre juegos de mesa, otro con datos sobre películas y, además, se recopila información sobre libros de una fuente externa. También se incluye un conjunto de palabras obtenido de Kaggle, el cual permite que el algoritmo evalúe el estado de ánimo del usuario (alegre o triste) y ajuste la recomendación en consecuencia. Este enfoque permite que el sistema sugiera actividades que se alineen con el estado de ánimo y las preferencias del usuario, con el fin de ofrecer una experiencia más satisfactoria.

Introducción

Con el avance de la tecnología y el aumento de los datos disponibles en diversas plataformas, la personalización del contenido se ha convertido en un factor esencial para mejorar la experiencia del usuario. En particular, los sistemas de recomendación desempeñan un papel clave al ofrecer opciones personalizadas, adaptándose a los gustos y emociones de cada persona. El objetivo de este proyecto es desarrollar un asistente de recomendaciones de entretenimiento que, basándose en el estado de ánimo y las preferencias del usuario, pueda sugerir películas, libros o juegos de mesa. El sistema no solo tiene en cuenta los intereses explícitos del usuario, sino que también realiza un análisis de sentimiento a partir de las palabras empleadas. La implementación de este tipo de algoritmos es útil en aplicaciones de entretenimiento, ya que facilitan la elección de actividades de ocio, especialmente en circunstancias en las que las opciones pueden estar limitadas, como en días de lluvia.

Objetivos específicos

Desarrollar un sistema de procesamiento del lenguaje natural para identificar el estado de ánimo del usuario a partir de un análisis de sentimiento de sus entradas.

Implementar un sistema de recomendación basado en machine learning que sugiera películas, libros o juegos de mesa según las preferencias y emociones detectadas en el usuario.

Integrar y preprocesar diversos conjuntos de datos (películas, juegos y palabras positivas/negativas) para permitir un análisis eficaz y una recomendación precisa.

Fuentes de datos

Los datos con los que trabajamos incluyen dos archivos CSV que contienen información detallada sobre películas y juegos de mesa.

El archivo *"IMDB-Movie-Data.csv"* proporciona datos sobre películas, incluyendo:

- El ranking de cada película ("Rank"),
- El título ("Title")
- El género ("Genre")
- Una breve descripción ("Description")
- El director ("Director")
- Actores ("Actors")
- El año de lanzamiento ("Year")
- La duración en minutos ("Runtime (Minutes)")
- La puntuación promedio de los usuarios ("Rating")
- El número de votos recibidos ("Votes")
- Los ingresos generados en millones ("Revenue (Millions)")
- Una puntuación adicional ("Metascore").

El segundo archivo, *"bgg_database.csv"*, contiene información sobre juegos de mesa. Entre los datos disponibles se encuentran:

- El ranking de los juegos ("rank"), }
- El nombre del juego ("game_name"),
- Un enlace a una página con una descripción completa e imágenes del juego ("game_href"),
- Clasificaciones en un portal de juegos, como el "geek_rating" y el "avg_rating".
- Incluye el número de votos recibidos ("num_voters"),
- Una descripción del juego ("description"),
- El año de publicación ("yearpublished"),
- El número mínimo y máximo de jugadores ("minplayers" y "maxplayers"),

- El tiempo mínimo y máximo para completar una partida (“minplaytime” y “maxplaytime”),
- La edad mínima recomendada para los jugadores (“minage”),
- El peso promedio del juego, que indica su complejidad (“avgweight”),
- La mejor cantidad de jugadores para disfrutar del juego (“best_num_players”),
- Los diseñadores (“mechanics”), y tanto la mecánica como la categoría del juego (“categories”).

Se extrajeron los datos de los libros de la página Proyecto Gutenberg, donde se recopiló la información de cada obra introduciéndolos en el HTML de la página para identificar con precisión la ubicación de los datos. Para cada libro, recopilamos

- Título (“Título”),
- Autor (“Autor”),
- Un enlace para ver más detalles y descargar el libro (“Enlace”),
- Un breve resumen de la obra (“Resumen”),
- Los temas relacionados con el autor (“Subjects”).

Esta información fue almacenada en un archivo CSV llamado “libros.csv” para su posterior procesamiento y análisis.

Por último, se obtuvieron dos archivos de texto con palabras clasificadas en positivo y negativo, denominados “*positive-words.txt*” y “*negative-words.txt*”. Ambos archivos están en inglés y contienen términos comunes para analizar el sentimiento en las entradas de los usuarios. Estos archivos fueron descargados de una fuente pública en Kaggle, especializada en el procesamiento de lenguaje natural y análisis de sentimientos.

Desarrollo

El proceso de desarrollo del ChatBot pasó por varias etapas. Inicialmente, se configuró un preprocesamiento de texto utilizando un objeto en Python que limpia y normaliza el texto, eliminando puntuaciones, convirtiéndolo a minúsculas y eliminando palabras vacías. Esta etapa permitió preparar el texto para un análisis de sentimientos más preciso y para la recomendación de contenido.

A continuación, se diseñó una función para capturar la entrada del usuario. Se optó por una estructura en la que el usuario pudiera indicar sus emociones e intereses en una sola línea, separando ambas partes con comas. Esta entrada se tradujo al inglés para garantizar la compatibilidad con los modelos de análisis de sentimientos, utilizando Google Translator, y luego se limpió con el preprocesador previamente creado.

Se implementaron funciones para el análisis de sentimientos y la generación de recomendaciones de contenido. Para la clasificación de sentimientos, se desarrolló un modelo de regresión logística que evalúa palabras positivas y negativas para interpretar el tono emocional en la entrada del usuario. En cuanto al sistema de recomendaciones, se construyó un pipeline en dos etapas: primero, se utilizó una regresión logística para identificar la intención del usuario, y luego, se aplicó un vectorizador que, mediante la similitud del coseno, selecciona una respuesta adecuada a partir de un conjunto de datos, basada en el tema que el usuario ha mencionado.

Finalmente, todas las funciones se unieron en una estructura que permite al chatbot responder en tiempo real, indicar al usuario su estado emocional y sugerir contenido relevante. Este proceso implicó la integración de bibliotecas de procesamiento de lenguaje natural, así como técnicas de formación de modelos en un flujo automatizado para lograr una experiencia de chatbot interactiva y fluida

Implementación

Preprocesado

Clase TextPreprocessor

La clase *TextPreprocessor* es un objeto personalizado que se utiliza para preprocesar texto antes de aplicarlo a nuestros modelos. Esta clase hereda de *BaseEstimator* y *TransformerMixin*.

La clase tiene como objetivo limpiar y preprocesar el texto, eliminando stopwords, puntuación y convirtiendo el texto a minúsculas. Esto es útil para normalizar el texto antes de convertirlo en características numéricas (por ejemplo, mediante TF-IDF).

```
class TextPreprocessor(BaseEstimator, TransformerMixin):
    """
    TextPreprocessor es una clase de preprocesamiento de texto para limpiar datos textuales.
    Hereda de BaseEstimator y TransformerMixin de scikit-learn para integrarse en pipelines
    y garantizar compatibilidad con la API de scikit-learn.
    """

    def __init__(self):
        # Conjunto de stopwords en inglés que serán eliminadas del texto
        self.stop_words = set(stopwords.words('english'))

    def clean_text(self, text: str) -> str:
        """
```

Limpia el texto de entrada realizando los siguientes pasos:

1. Convierte el texto a minúsculas.
2. Elimina la puntuación.
3. Elimina las stopwords.

Parámetros:

text (str): Texto de entrada a limpiar.

Retorna:

str: Texto limpio.

"""

Convertir el texto a minúsculas

text = text.lower()

Eliminar puntuación del texto

text = text.translate(str.maketrans("", "", string.punctuation))

Eliminar stopwords filtrando palabras en el texto

text = ' '.join(word for word in text.split() if word not in self.stop_words)

return text

def fit(self, X: pd.Series, y=None) -> "TextPreprocessor":

"""

Método de ajuste para cumplir con la API de scikit-learn.

Este transformador no necesita aprender nada de los datos.

Parámetros:

X (pd.Series): Serie de pandas con los datos textuales de entrada.

y: Parámetro opcional para compatibilidad; no se utiliza.

Retorna:

TextPreprocessor: Retorna la instancia actual del objeto.

"""

return self

def transform(self, X: pd.Series) -> pd.Series:

"""

Aplica el proceso de limpieza de texto a los datos de entrada.

Parámetros:

X (pd.Series o str): Datos de entrada a transformar, ya sea una Serie de pandas o una cadena individual.

Retorna:

pd.Series o str: Datos limpiados, retornados como una Serie de pandas si la entrada fue una Serie,

o como una cadena individual si la entrada fue una cadena.

Excepciones:

ValueError: Si el tipo de entrada no es ni una Serie de pandas ni una cadena.

"""

Verificar si la entrada es una Serie de pandas

```
if isinstance(X, pd.Series):  
    return X.apply(self.clean_text)  
# Verificar si la entrada es una cadena individual  
elif isinstance(X, str):  
    return self.clean_text(X)  
# Lanzar un error si el tipo de entrada es inválido  
else:  
    raise ValueError("La entrada debe ser una cadena o una Serie de pandas.")
```

`__init__`: Se inicializa un conjunto de stopwords usando la lista predeterminada en `nltk.corpus.stopwords` para el idioma inglés. Las stopwords son palabras que generalmente no aportan valor significativo al análisis de texto (por ejemplo, "y", "el", "de").

`clean_text`: Este es el método principal de preprocesamiento:

- Convierte el texto a minúsculas usando `text.lower()`.
- Elimina la puntuación utilizando `str.translate` con `string.punctuation` para eliminar los caracteres de puntuación comunes.
- Elimina las stopwords mediante una comprensión de listas que conserva solo las palabras que no están en el conjunto de `stop_words`.
- Este método devuelve el texto limpio y procesado.

`fit`: Este método es requerido por el `TransformerMixin` de `scikit-learn`. En este caso, el preprocesador no necesita hacer un ajuste a los datos, por lo que simplemente devuelve el objeto `self`. Esto permite que la clase sea utilizada en un pipeline.

`transform`: Este método aplica el preprocesamiento al texto:

- Si la entrada es un `pandas.Series` (una columna de texto en un `DataFrame`), aplica `clean_text` a cada elemento de la serie utilizando `apply`.
- Si la entrada es un único string, se aplica directamente el método `clean_text` al string.
- Si la entrada no es ni un `pandas.Series` ni un string, se lanza un `ValueError`.

Carga

Función `load_books`

La función `load_books()` hace web scraping de la página de Project Gutenberg, extrayendo títulos, autores, resúmenes y temas de los libros más populares. La información se guarda en un archivo CSV. Si se activa el `developer_mode`, imprime detalles de los libros mientras realiza el scraping.


```
def load_books(developer_mode: bool = False) -> None:
    """
    Descarga información de libros populares del sitio Proyecto Gutenberg.
    Extrae el título, autor, enlace, resumen y temas de cada libro, y guarda los datos en un
    archivo CSV.

    Parámetros:
    developer_mode (bool): Modo desarrollador. Si es True, imprime información detallada de
    cada libro mientras se procesa.

    Salida:
    None: La función guarda los datos en un archivo CSV y no retorna ningún valor.
    """

    # URL de la página de libros más populares
    url: str = "https://www.gutenberg.org/browse/scores/top1000.php#books-last1"

    # Realizar solicitud a la página principal de libros
    response = requests.get(url)
    response.raise_for_status()
    soup = BeautifulSoup(response.text, 'html.parser')

    # Seleccionar enlaces de libros que contienen '/ebooks/' seguido de un número (ID de libro)
    libros = soup.select("li > a[href^='/ebooks/']")
    datos_libros: List[Dict[str, str]] = [] # Lista para almacenar la información de cada libro

    # Procesar cada enlace de libro
    for libro in libros:
        texto_completo: str = libro.get_text()
        enlace_anidado: str = libro['href']

        # Verificar que el enlace tenga un número después de "/ebooks/"
        if enlace_anidado.split('/ebooks/')[1].isdigit():
            # Separar título y autor; si no hay autor, se asigna "Desconocido"
            titulo_y_autor: List[str] = texto_completo.split(" by ", 1)
            titulo: str = titulo_y_autor[0].strip()
            autor: str = titulo_y_autor[1].strip() if len(titulo_y_autor) > 1 else "Desconocido"

            # Construir URL completa para la página del libro
            url_libro: str = f"https://www.gutenberg.org{enlace_anidado}"

            # Realizar solicitud a la página individual del libro
            response_libro = requests.get(url_libro)
            response_libro.raise_for_status()
            soup_libro = BeautifulSoup(response_libro.text, 'html.parser')

            # Intentar extraer el resumen si está disponible
            summary: str = ""
```

```
summary_row = soup_libro.find('th', text='Summary')
if summary_row:
    summary_td = summary_row.find_next("td")
    if summary_td:
        summary = summary_td.get_text(strip=True)

# Extraer temas (subjects) del libro
subjects: List[str] = []
subject_rows = soup_libro.find_all('th', text='Subject')
for subject_row in subject_rows:
    subject_td = subject_row.find_next("td")
    if subject_td:
        subject_links = subject_td.find_all('a')
        for link in subject_links:
            subjects.append(link.get_text(strip=True))

# Combinar los temas en una sola cadena
subjects_combined: str = ", ".join(subjects)

# Añadir la información del libro a la lista de datos
datos_libros.append({
    "Título": titulo,
    "Autor": autor,
    "Enlace": url_libro,
    "Resumen": summary,
    "Subjects": subjects_combined
})

# Imprimir información detallada en modo desarrollador
if developer_mode:
    print("Título:", titulo, "Autor:", autor, "URL:", url_libro, "Resumen:", summary, "Temas:",
subjects_combined)

# Crear un DataFrame y guardar los datos en un archivo CSV
df_libros = pd.DataFrame(datos_libros)
df_libros.to_csv('libros.csv', index=False)
```

La función realiza una solicitud HTTP mediante *requests.get(url)* para obtener el contenido de la página que contiene la lista de los 1000 libros más populares de Project Gutenberg. La llamada *raise_for_status()* garantiza que, si la solicitud falla (por ejemplo, si el servidor devuelve un error 404), se lance una excepción para notificar el problema.

Una vez obtenida la respuesta, el contenido HTML de la página es procesado usando BeautifulSoup. Para extraer los enlaces a los libros, se utilizan selectores CSS (*soup.select(...)*). La expresión

a[href^='/ebooks/']

Selecciona todos los enlaces cuyo atributo href empieza con '/ebooks/', lo que indica que estos enlaces corresponden a libros disponibles para su descarga.

Para cada enlace de libro, se realiza lo siguiente:

- *libro.get_text()* extrae el texto visible en el enlace, que típicamente contiene el título y el autor del libro.
- *libro['href']* obtiene la URL relativa del libro.

A continuación, se verifica que el enlace contenga un número después de la parte "/ebooks/", lo cual asegura que el enlace esté relacionado con un libro específico y no con otros contenidos de la página.

El texto extraído del enlace se separa en dos partes utilizando el separador " by ": una para el título y otra para el autor. Si el autor no está disponible, se asigna el valor "Desconocido". Después, se genera la URL completa para acceder a la página del libro mediante el enlace relativo.

Se realiza una nueva solicitud para acceder a la página individual de cada libro y se procesa el HTML de la misma manera con BeautifulSoup. Se busca el resumen del libro, localizando el encabezado "Summary" y extrayendo el texto correspondiente. También se buscan todos los "Subjects" (temas) del libro, extrayendo los enlaces dentro de la sección de temas y recopilando sus textos. Los temas extraídos se combinan en una sola cadena, separados por comas.

Una vez obtenidos los datos (título, autor, resumen y temas), se almacenan en un diccionario y se agregan a la lista *datos_libros* que luego se guardan en un archivo CSV llamado libros.csv.

Función load_words

La función *load_words()* recibe como argumento la ruta del archivo que contiene las palabras (en nuestro caso, positive-words.txt y negative-words.txt) y devuelve un set con las palabras.

```
def load_words(file_path: str) -> set[str]:  
    """  
    Carga palabras desde un archivo y las retorna como un conjunto de cadenas.  
  
    Parámetros:  
    file_path (str): La ruta del archivo de texto que contiene una palabra por línea.  
  
    Retorna:  
    Set[str]: Un conjunto de palabras (strings) obtenidas del archivo, sin espacios en blanco.  
    """
```

```
with open(file_path, 'r') as file:
    # Lee todas las líneas del archivo y elimina los espacios en blanco alrededor de cada
    palabra
    words = {line.strip() for line in file.readlines()}

return words
```

Función load_datasets_union

La función `load_datasets_union()` carga y procesa tres conjuntos de datos distintos: películas, juegos de mesa y libros. Para cada conjunto, la función realiza un preprocesamiento de texto, renombrar columnas para unificar los nombres y concatenar todos los datos en un solo DataFrame, agregando una nueva columna de texto procesado que combina varias características de cada conjunto. Esta función tiene como objetivo crear un único DataFrame consolidado con datos de diferentes categorías, listo para ser utilizado en análisis o modelos de procesamiento de lenguaje natural.

```
def load_datasets_union() -> pd.DataFrame:
    """
    Carga y preprocesa tres conjuntos de datos (películas, juegos de mesa y libros),
    limpiando el texto de las columnas relevantes y combinándolos en un solo DataFrame.

    Realiza las siguientes acciones:
    1. Procesa y limpia los datos de películas.
    2. Procesa y limpia los datos de juegos de mesa.
    3. Procesa y limpia los datos de libros.
    4. Combina todos los datos en un solo DataFrame.

    Retorna:
    pd.DataFrame: Un DataFrame combinado que contiene los datos preprocesados.
    """
    # Instanciar el preprocesador de texto
    text_preprocessor = TextPreprocessor()

    # 1. Procesamiento de Películas
    peliculas_dataframe = pd.read_csv('database/IMDB-Movie-Data.csv')
    peliculas_dataframe['category'] = 'pelicula'
    peliculas_dataframe['text'] = (
        peliculas_dataframe['Title'].fillna("").apply(text_preprocessor.transform) + " " +
        peliculas_dataframe['Genre'].fillna("").apply(text_preprocessor.transform) + " " +
        peliculas_dataframe['Description'].fillna("").apply(text_preprocessor.transform) + " " +
        peliculas_dataframe['Director'].fillna("").apply(text_preprocessor.transform) + " " +
        peliculas_dataframe['Actors'].fillna("").apply(text_preprocessor.transform)
    )
    peliculas_dataframe.rename(columns={'Title': 'Titulo', 'Director': 'Autor', 'Description':
'Resumen', 'Genre': 'Subjects'}, inplace=True)
```

```
# 2. Procesamiento de Juegos de Mesa
juegos_mesa_dataframe = pd.read_csv('database/bgg_database.csv')
juegos_mesa_dataframe['category'] = 'juego'
juegos_mesa_dataframe['text'] = (
    juegos_mesa_dataframe['game_name'].fillna("").apply(text_preprocessor.transform) + " " +
    juegos_mesa_dataframe['description'].fillna("").apply(text_preprocessor.transform) + " " +
    juegos_mesa_dataframe['designers'].apply(lambda x: ' '.join(eval(x)) if isinstance(x, str) else
    "").apply(text_preprocessor.transform).fillna("") + " " +
    juegos_mesa_dataframe['mechanics'].apply(lambda x: ' '.join(eval(x)) if isinstance(x, str)
    else "").apply(text_preprocessor.transform).fillna("") + " " +
    juegos_mesa_dataframe['categories'].apply(lambda x: ' '.join(eval(x)) if isinstance(x, str)
    else "").apply(text_preprocessor.transform).fillna("")
)
juegos_mesa_dataframe.rename(columns={'game_name': 'Titulo', 'game_href': 'Enlace',
'designers': 'Autor', 'description': 'Resumen', 'mechanics': 'Subjects'}, inplace=True)

# 3. Procesamiento de Libros
libros_dataframe = pd.read_csv('database/libros.csv')
libros_dataframe['category'] = 'libro'
libros_dataframe['text'] = (
    libros_dataframe['Título'].fillna("").apply(text_preprocessor.transform) + " " +
    libros_dataframe['Resumen'].fillna("").apply(text_preprocessor.transform) + " " +
    libros_dataframe['Subjects'].fillna("").apply(text_preprocessor.transform) + " " +
    libros_dataframe['Autor'].fillna("").apply(text_preprocessor.transform)
)
libros_dataframe.rename(columns={'Título': 'Titulo', 'Autor': 'Autor', 'Resumen': 'Resumen',
'Subjects': 'Subjects'}, inplace=True)

# Concatenar los DataFrames de películas, juegos de mesa y libros en un solo DataFrame
dataframe_bbdd = pd.concat(
    [peliculas_dataframe[['Titulo', 'Autor', 'Resumen', 'Subjects', 'category', 'text']],
    juegos_mesa_dataframe[['Titulo', 'Enlace', 'Autor', 'Resumen', 'Subjects', 'category', 'text']],
    libros_dataframe[['Titulo', 'Enlace', 'Autor', 'Resumen', 'Subjects', 'category', 'text']]
    ],
    ignore_index=True # Para reiniciar el índice después de la concatenación
)

return dataframe_bbdd
```

Se crea una instancia de la clase *TextPreprocessor*. Este objeto es responsable de limpiar y preprocesar el texto (como convertir a minúsculas, eliminar stopwords, etc.), y se utilizará para procesar las columnas de texto en cada uno de los conjuntos de datos.

Se leen los archivos CSV que contiene la base de datos y se asignan nombres de columna más legibles. Luego, se genera una nueva columna *text*, que es la concatenación de varias características. Cada columna se procesa mediante el preprocesador de texto (*text_preprocessor.transform()*), y los valores nulos se manejan con *.fillna("")*.

Los tres DataFrames procesados (películas, juegos de mesa y libros) se concatenan en un solo DataFrame llamado *dataframe_bbdd*. Se seleccionan las columnas relevantes de cada DataFrame antes de la concatenación.

Finalmente, la función devuelve el DataFrame resultante que contiene los datos de todas las categorías de manera unificada

Función `count_words`

La función `count_words()` está diseñada para contar cuántas palabras positivas y negativas aparecen en un texto dado, utilizando listas predefinidas de palabras positivas y negativas.

La función recibe tres argumentos:

- *text*: El texto en el que se buscarán las palabras.
- *positive_words*: Un conjunto o lista de palabras consideradas positivas.
- *negative_words*: Un conjunto o lista de palabras consideradas negativas.

El propósito de la función es contar cuántas de las palabras del texto están presentes en las listas de palabras positivas y negativas, y devolver una serie de pandas.

```
def count_words(text: str, positive_words: set[str], negative_words: set[str]) -> pd.Series:
    """
    Cuenta las palabras positivas y negativas en un texto dado.

    Esta función cuenta cuántas palabras del texto pertenecen al conjunto de
    palabras positivas y cuántas al conjunto de palabras negativas.

    Parámetros:
    - text (str): El texto en el cual se realizarán las búsquedas de palabras.
    - positive_words (Set[str]): Un conjunto de palabras consideradas positivas.
    - negative_words (Set[str]): Un conjunto de palabras consideradas negativas.

    Retorna:
    - pd.Series: Una serie de pandas con dos valores, el conteo de palabras positivas
      en el texto y el conteo de palabras negativas en el texto, en ese orden.
    """
    # Contar las palabras positivas en el texto
    positive_count = sum(1 for word in text.split() if word in positive_words)

    # Contar las palabras negativas en el texto
    negative_count = sum(1 for word in text.split() if word in negative_words)

    # Retornar los resultados como una Serie de pandas
```

```
return pd.Series([positive_count, negative_count], index=['Positive_Count', 'Negative_Count'])
```

Función `procesar_entrada_usuario`:

La función `procesar_entrada_usuario()` está diseñada para procesar una entrada de texto en español proporcionada por el usuario. El proceso incluye la traducción del texto al inglés y luego su preprocesamiento con una instancia de la clase `TextPreprocessor` para una posterior utilización en los modelos.

```
def procesar_entrada_usuario(texto) -> str:
    """
    Procesa la entrada del usuario de la siguiente manera:
    1. Traduce el texto de español a inglés.
    2. Preprocesa el texto traducido para limpieza (ej. eliminación de stopwords, caracteres no
    deseados).

    Parámetros:
    - texto (str): Texto en español ingresado por el usuario.

    Retorna:
    - texto_procesado (str): Texto limpio y procesado en inglés.
    """
    # Traducir al inglés
    texto_traducido = GoogleTranslator(source='es', target='en').translate(texto)

    # Preprocesar el texto traducido (limpieza de texto)
    texto_procesado = TextPreprocessor().clean_text(texto_traducido)

    return texto_procesado
```

Modelos

Función `model_sentiments`

La función entrena un modelo de clasificación para predecir si una palabra pertenece a un sentimiento "feliz" o "triste" utilizando regresión logística. Se utiliza este modelo para clasificar las palabras en dos categorías (sentimiento positivo o negativo). Es un modelo de clasificación binaria que predice la probabilidad de una clase dada una serie de características.

Palabras positivas y negativas: La función carga dos archivos de texto (`positive-words.txt` y `negative-words.txt`) que contienen listas de palabras etiquetadas como "positivas" o "negativas".

Etiquetas: Las palabras positivas se etiquetan como "feliz" y las palabras negativas como "triste". Esto se usa para crear un conjunto de datos etiquetado.

Para cada palabra, se cuentan las instancias de palabras positivas y negativas en el texto. Las etiquetas de sentimientos ("feliz" y "triste") se convierten en valores numéricos (1 para "feliz" y 0 para "triste") para ser utilizadas en el modelo de machine learning.

Se retorna el modelo entrenado junto con las listas de palabras positivas y negativas son devueltos como resultado, permitiendo que se pueda usar para hacer predicciones sobre nuevos textos.

```
def model_sentiments(developer_mode: bool) -> tuple[LogisticRegression, set[str], set[str]]:
    """
    Entrena un modelo de regresión logística para predecir sentimientos (positivo o negativo)
    basado en palabras clave positivas y negativas.

    La función carga un conjunto de palabras positivas y negativas desde archivos, las convierte
    en un DataFrame y genera
    características basadas en la frecuencia de aparición de palabras positivas y negativas en el
    texto. Luego entrena un modelo
    de regresión logística para clasificar las palabras como "feliz" (positivo) o "triste" (negativo).

    Parámetros:
    - developer_mode (bool): Un indicador para mostrar las métricas de evaluación del modelo
    (accuracy, matriz de confusión, etc.).

    Retorna:
    - model (LogisticRegression): El modelo entrenado de regresión logística.
    - positive_words (Set[str]): El conjunto de palabras positivas cargadas.
    - negative_words (Set[str]): El conjunto de palabras negativas cargadas.
    """

    # Cargar palabras positivas y negativas
    positive_words = load_words('database/positive-words.txt')
    negative_words = load_words('database/negative-words.txt')

    # Crear un conjunto de datos basado en las palabras
    data = {
        'text': [],
        'label': []
    }

    # Llenar el conjunto de datos con palabras positivas y negativas
    for word in positive_words:
        data['text'].append(word)
        data['label'].append('feliz') # Etiqueta "feliz" para palabras positivas

    for word in negative_words:
        data['text'].append(word)
        data['label'].append('triste') # Etiqueta "triste" para palabras negativas
```



```
# Crear un DataFrame
df = pd.DataFrame(data)

# Crear características
df[['positive_count', 'negative_count']] = df['text'].apply(lambda x: count_words(x,
positive_words, negative_words))

# Convertir etiquetas a números: "feliz" -> 1, "triste" -> 0
df['label'] = df['label'].map({'feliz': 1, 'triste': 0})

# Separar características y etiquetas
X = df[['positive_count', 'negative_count']] # Características
y = df['label'] # Etiquetas

# Dividir el conjunto de datos en entrenamiento y prueba (60% entrenamiento, 40% prueba)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

# Entrenar el modelo de regresión logística
model = LogisticRegression()
model.fit(X_train, y_train)

if developer_mode:
    # Predecir en el conjunto de prueba
    y_pred = model.predict(X_test)

    # Evaluar el modelo
    accuracy = accuracy_score(y_test, y_pred)
    confusion = confusion_matrix(y_test, y_pred)
    report = classification_report(y_test, y_pred)

    print(f'Accuracy: {accuracy}')
    print('Confusion Matrix:')
    print(confusion)
    print('Classification Report:')
    print(report)

# Retornar el modelo entrenado y los conjuntos de palabras positivas y negativas
return model, positive_words, negative_words
```

Si el parámetro `developer_mode` está activado, el modelo se evalúa en el conjunto de prueba mediante:

- Precisión (Accuracy): Mide la proporción de predicciones correctas.
- Matriz de confusión: Muestra los resultados de la clasificación, como los verdaderos positivos, verdaderos negativos, etc.
- Reporte de clasificación: Proporciona métricas como precisión, recall y f1-score del modelo entrenado.

Función train_model

La función entrena un modelo de clasificación basado en texto utilizando TF-IDF (Term Frequency-Inverse Document Frequency) para convertir el texto en características numéricas y un modelo de regresión logística para la clasificación.

El modelo está diseñado para clasificar la intención del usuario entre tres categorías predefinidas (libros, películas y juegos de mesa), basándose en el texto proporcionado por el usuario. Una vez entrenado, el modelo puede predecir la categoría (intención) del texto ingresado, lo cual es el primer paso para proporcionar una recomendación dentro de esa categoría.

```
def train_model(dataset: pd.DataFrame, developer_mode: bool) -> any:
    """
    Entrena un modelo de clasificación de texto utilizando regresión logística y TF-IDF para
    vectorización de texto.

    La función divide el conjunto de datos en entrenamiento y prueba, luego entrena un modelo
    con regresión logística y realiza
    una validación cruzada para evaluar el rendimiento del modelo. Si 'developer_mode' es True,
    imprime las puntuaciones de la
    validación cruzada y la puntuación media.

    Parámetros:
    - dataset (pd.DataFrame): Un DataFrame con columnas 'text' (texto) y 'category' (categorías
    de las clases).
    - developer_mode (bool): Un indicador para imprimir las métricas de la validación cruzada si
    es True.

    Retorna:
    - model (Pipeline): El modelo entrenado con regresión logística y vectorización TF-IDF.
    """

    # Separar el dataset en conjunto de entrenamiento y conjunto de prueba (80% - 20%)
    X_train, X_test, y_train, y_test = train_test_split(dataset["text"], dataset["category"],
    test_size=0.2, random_state=42)

    # Crear un pipeline con TF-IDF para la vectorización de texto y regresión logística
    model = make_pipeline(
        TfidfVectorizer(), # Vectoriza el texto con TF-IDF
        LogisticRegression(C=1.0, max_iter=200, solver='liblinear') # Modelo de regresión logística
    )

    # Entrenar el modelo
    model.fit(X_train, y_train)
```

```
# Realizar validación cruzada (5 pliegues)
cv_scores = cross_val_score(model, X_train, y_train, cv=5) # 5-fold cross-validation

# Si 'developer_mode' es True, imprimir detalles de la validación cruzada
if developer_mode:
    print(f"Cross-validation scores: {cv_scores}")
    print(f"Mean cross-validation score: {cv_scores.mean()}")

return model
```

El modelo de regresión logística se usa para clasificar los textos en categorías. Este es un modelo lineal utilizado comúnmente para clasificación binaria o multiclase.

Se configura con hiperparámetros específicos:

- *C=1.0*: Es el parámetro de regularización que controla el equilibrio entre la precisión del modelo y la complejidad.
- *max_iter=200*: Define el número máximo de iteraciones para el proceso de optimización.
- *solver='liblinear'*: Es el algoritmo utilizado para la optimización, adecuado para conjuntos de datos pequeños o medianos.

Validación cruzada: Se utiliza la función *cross_val_score* para realizar una validación cruzada de 5 pliegues (5-fold cross-validation) sobre los datos de entrenamiento. Esto permite obtener una evaluación más robusta del modelo.

Puntuaciones de validación cruzada: Se imprimen las puntuaciones de cada pliegue y el promedio de estas puntuaciones si *developer_mode* es True. Esto es útil para evaluar la estabilidad y el rendimiento del modelo.

La función devuelve el modelo entrenado, que se puede usar para hacer predicciones sobre nuevos datos.

Función *model_recommender*

La función *model_recommender()* tiene como objetivo vectorizar el texto de un conjunto de datos utilizando el método TF-IDF, el cual convierte los textos en vectores numéricos que pueden ser procesados por los modelos.

Retorna la instancia del vectorizer, y la matriz generada.

```
def model_recommender(dataset) -> any:
    """
    Esta función vectoriza los textos del dataset utilizando el algoritmo TF-IDF (Term
    Frequency-Inverse Document Frequency)
    para representar los textos como vectores numéricos.
```

Parámetros:

- dataset (pd.DataFrame): El DataFrame que contiene la columna 'text', que contiene los textos a vectorizar.

Retorna:

- tfidf_matrix (scipy.sparse.csr_matrix): Matriz dispersa de características vectorizadas del texto.
- tfidf_vectorizer (TfidfVectorizer): El objeto del vectorizador que puede usarse para transformar más texto.

"""

Crear un objeto TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()

Ajustar y transformar los textos en una matriz TF-IDF

tfidf_matrix = tfidf_vectorizer.fit_transform(dataset['text'])

return tfidf_matrix, tfidf_vectorizer

Predicciones

Función recommend_item

La función se utiliza para clasificar un texto de entrada (en este caso, el "prompt") en una de las categorías predefinidas que el modelo ha aprendido a distinguir (libros, películas o juegos de mesa). Este es el paso en el proceso de clasificación basado en la intención del usuario, y una vez que se determina la categoría, el sistema puede proceder a recomendar elementos dentro de esa categoría.

```
def recommend_item(prompt, model) -> str:
```

"""

Esta función recomienda un artículo basado en el prompt del usuario.

El modelo predictivo se usa para predecir la categoría del texto de entrada.

Parámetros:

- prompt (str): El texto de entrada del usuario.

- model (modelo entrenado): El modelo que se usará para predecir la categoría.

Retorna:

- category (str): La categoría predicha por el modelo para el prompt ingresado.

"""

Predecir la categoría del prompt

category = model.predict([prompt])[0]

```
return category
```

Función predict_sentiments

La función predict_sentiments() está diseñada para predecir la emoción de un texto (frase) de entrada utilizando un modelo previamente entrenado. Esta función evalúa si el sentimiento de una frase es "feliz" o "triste" en función de las palabras positivas y negativas contenidas en ella, utilizando un modelo de clasificación. El modelo utiliza las frecuencias de palabras positivas y negativas para clasificar el texto. La predicción es un número (0 o 1), donde 1 indica "feliz" y 0 indica "triste".

```
def predict_sentiments(model, positive_words, negative_words, frase, developer_mode) -> str:
    """
    Esta función predice la emoción (feliz o triste) en función de la frase dada.
    La predicción se realiza contando las palabras positivas y negativas en la frase procesada
    y utilizando un modelo previamente entrenado.

    Parámetros:
    - model (LogisticRegression o similar): El modelo entrenado para predecir la emoción.
    - positive_words (list): Lista de palabras positivas.
    - negative_words (list): Lista de palabras negativas.
    - frase (str): La entrada de texto a procesar y analizar.
    - developer_mode (bool): Indica si se deben imprimir detalles adicionales para depuración.

    Retorna:
    - emocion (str): La emoción predicha, que puede ser 'feliz' o 'triste'.
    """

    # Procesar la entrada de texto (traducción y limpieza)
    entrada_procesada = procesar_entrada_usuario(frase)

    if developer_mode:
        print("Texto procesado:", entrada_procesada)

    # Contar las palabras positivas y negativas en la entrada procesada
    entrada_counts = count_words(entrada_procesada, positive_words, negative_words)

    # Asegurarse de que la entrada esté en formato 2D para la predicción
    entrada_vectorizada = np.array([entrada_counts]).reshape(1, -1)

    # Obtener la predicción del modelo: 1 para 'feliz', 0 para 'triste'
    prediccion = model.predict(entrada_vectorizada)[0]

    # Determinar la emoción basada en la predicción (1 -> feliz, 0 -> triste)
    emocion = 'feliz' if prediccion == 1 else 'triste'

    return emocion
```

Función `predict_recomendation`

La función `predict_recomendation()` tiene como objetivo hacer una recomendación basada en el prompt proporcionado y la categoría que se pasa como parámetro. Utiliza un modelo de regresión logística (que previamente se entrena con un `TfidfVectorizer`), y calcula la similitud entre el prompt y los elementos en la base de datos para encontrar la recomendación más adecuada.

```
def predict_recomendation(prompt, category, model, dataframe_bbdd) -> tuple[str, str]:  
    """  
    Esta función genera recomendaciones basadas en un 'prompt' de entrada,  
    filtrando las recomendaciones según la categoría dada, y luego calculando  
    la similitud de coseno entre el 'prompt' y los elementos del conjunto de datos.  
  
    Parámetros:  
    - prompt (str): El texto de entrada para la recomendación.  
    - category (str): La categoría de recomendaciones a considerar.  
    - model (Pipeline): El modelo entrenado que incluye el vectorizador TF-IDF.  
    - dataframe_bbdd (DataFrame): El conjunto de datos de recomendaciones, con columnas  
    'category', 'text', 'Titulo', 'Enlace'.  
  
    Retorna:  
    - Titulo (str): El título de la recomendación más similar al 'prompt'.  
    - Enlace (str): El enlace relacionado con la recomendación.  
    """  
  
    # Filtrar el dataframe_bbdd para obtener solo las recomendaciones de la categoría dada.  
    recommendations = dataframe_bbdd[dataframe_bbdd['category'] ==  
category].copy().reset_index(drop=True)  
  
    # Si no hay recomendaciones disponibles para la categoría, devolver un mensaje.  
    if recommendations.empty:  
        return "No hay recomendaciones disponibles", ""  
  
    # Vectorizar el 'prompt' de recomendación utilizando el vectorizador TF-IDF del modelo.  
    prompt_vector = model.named_steps["tfidfvectorizer"].transform([prompt])  
  
    # Vectorizar el texto de las recomendaciones del dataset usando el vectorizador TF-IDF del  
    modelo.  
    dataset_vector = model.named_steps["tfidfvectorizer"].transform(recommendations["text"])  
  
    # Calcular la similitud de coseno entre el 'prompt' y el texto de las recomendaciones del  
    dataset.  
    similarity_scores = cosine_similarity(prompt_vector, dataset_vector)  
  
    # Obtener el índice del ítem más similar en el dataset basado en la similitud más alta.  
    most_similar_idx = similarity_scores.argmax()
```

```
# Obtener la recomendación correspondiente del dataframe_bbdd utilizando el índice de la  
similitud más alta.  
recommended_item = recommendations.iloc[most_similar_idx]  
  
# Devolver el título y el enlace de la recomendación más similar.  
return recommended_item['Titulo'], recommended_item['Enlace']
```

Se filtra el DataFrame para obtener solo las recomendaciones que corresponden a la categoría seleccionada. Esto asegura que las recomendaciones sean relevantes para el tipo de contenido solicitado.

El prompt proporcionado se transforma usando el *TfidfVectorizer* que está contenido en el modelo. Esto convierte el texto del prompt en un formato numérico que se puede comparar con el texto en el dataset.

Se calcula la similitud de coseno entre el vector del prompt y los vectores de texto en el dataset. La similitud de coseno mide cuán similares son dos vectores en un espacio multidimensional, siendo un valor cercano a 1 indicativo de alta similitud.

similarity_scores.argmax() encuentra el índice del ítem en el dataset con la mayor similitud al prompt. Este índice corresponde al ítem más similar al prompt.

Luego, se usa este índice para obtener el ítem recomendado del DataFrame *recommendations*. Se devuelve el título y el enlace del ítem recomendado.

Bot:

Función *input_user*

La función *input_user* está diseñada para recibir la entrada del usuario, procesar y devolver dos tipos de información: un prompt de sentimiento y uno de recomendación, que se extraen a partir del texto proporcionado.

La función solicita al usuario que ingrese un texto utilizando el comando *input()*. La entrada debe consistir en una oración que describa tanto cómo se siente el usuario como lo que le gustaría hacer. Ambos elementos deben separarse por una coma.

- La primera parte del texto (antes de la coma) corresponde al sentimiento del usuario.
- La segunda parte (después de la coma) se asigna a la recomendación o interés del usuario.

Por ejemplo, una entrada válida sería:

“Hoy tuve un buen día, quiero ver una película de criminales intergalácticos”.

Si el usuario escribe "Chau", la función termina y retorna "exit", indicando que desea finalizar la interacción.

Es importante respetar la estructura de la entrada:
“Cómo te sientes”, “Qué te interesa”.

```
def input_user():
    """
    Esta función solicita al usuario una entrada con dos partes:
    un sentimiento y una recomendación, separados por una coma.
    Si el usuario escribe "chau", la función termina la interacción.

    Retorna:
    - prompt_sentimiento (str): El texto procesado relacionado con el sentimiento.
    - prompt_recomendacion (str): El texto procesado relacionado con la recomendación, si
    existe.
    """

    # Solicitar la entrada del usuario
    prompt = input("¿Cómo estás hoy? ¿Qué quieres hacer? Puedes despedirte diciendo 'Chau':")

    # Verificar si el usuario quiere terminar la interacción
    if prompt.lower() == "chau":
        return "exit", "exit"

    # Separar el prompt en sentimientos y recomendaciones usando la coma como delimitador
    prompts = prompt.split(",")

    # Limpiar y asignar los textos a sus respectivas variables
    sentiment_prompt = prompts[0].strip() # El texto antes de la coma se considera el sentimiento
    recommendation_prompt = prompts[1].strip() if len(prompts) > 1 else "" # El texto después de
    la coma se considera la recomendación, si existe

    # Traducir los prompts de español a inglés usando Google Translator
    translated_sentiment = GoogleTranslator(source='es', target='en').translate(sentiment_prompt)
    translated_recommendation = GoogleTranslator(source='es',
    target='en').translate(recommendation_prompt)

    # Procesar el texto para eliminar caracteres no deseados y hacer limpieza
    prompt_sentimiento = TextPreprocessor().clean_text(translated_sentiment)
    prompt_recomendacion = TextPreprocessor().clean_text(translated_recommendation)

    # Retornar los prompts procesados
```



```
return prompt_sentimiento, prompt_recomendacion
```

Función print_slow

Esta función simula la escritura progresiva de ChatGPT, por cuestiones más estéticas que funcionales.

```
def print_slow(text, delay=0.04):  
    """  
    Imprime un texto lentamente, mostrando un carácter a la vez con un retraso especificado.  
  
    Parámetros:  
    - text (str): El texto que se imprimirá lentamente.  
    - delay (float, opcional): El tiempo de espera entre cada carácter en segundos.  
      Por defecto es 0.04 segundos.  
    """  
  
    # Recorrer cada carácter en el texto  
    for char in text:  
        sys.stdout.write(char) # Escribir el carácter en la salida estándar  
        sys.stdout.flush()    # Asegurarse de que el carácter se imprima inmediatamente  
        time.sleep(delay)     # Esperar antes de imprimir el siguiente carácter  
  
    # Imprimir una nueva línea al final  
    print()
```

Función main

La función comienza con la verificación de la presencia de los archivos que se utilizan como base de datos. Estos incluyen archivos de datos de juegos de mesa, películas, libros y las listas de palabras positivas y negativas utilizadas para el análisis de sentimientos. Si alguno de estos archivos no se encuentra disponible, se llama a las funciones pertinentes para realizar la carga de los mismos. Por ejemplo, si el archivo de libros no está presente, la función *load_books()* se encarga de cargar los datos correspondientes, mientras que la carpeta database se crea si no existe previamente.

Luego procede a entrenar los modelos necesarios para el sistema. Primero, se entrena el modelo de análisis de sentimientos mediante la función *model_sentiments()*. Además, se carga y fusiona la información de diferentes datasets en una estructura de datos unificada, utilizando la función

`load_datasets_union()`. Con esta base de datos combinada, el modelo de recomendaciones se entrena a través de la función `train_model`.

El programa entra en un bucle interactivo, en el que se solicita al usuario que ingrese un prompt válido. Este bucle continúa de manera indefinida hasta que el usuario ingresa el comando "Chau", lo que detiene la ejecución de la función.

Cada vez que el usuario ingresa un nuevo texto, el sistema realiza varias acciones: primero, predice el sentimiento utilizando el modelo entrenado de análisis de sentimientos; luego, determina la categoría más adecuada para la recomendación en función del texto ingresado; y finalmente, genera y muestra al usuario una recomendación personalizada, que incluye un título y un enlace relacionado con la categoría elegida.

```
def main(developer_mode) -> None:
    """
    Función principal que gestiona el flujo de trabajo de la aplicación.
    Verifica la existencia de archivos necesarios, entrena el modelo y
    maneja la interacción con el usuario para predecir sentimientos y
    hacer recomendaciones.

    Parámetros:
    - developer_mode (bool): Si está activado, muestra información detallada de depuración.
    """

    # Verificar la existencia de los archivos necesarios
    required_files = [
        'database/bgg_database.csv',
        'database/IMDB-Movie-Data.csv',
        'database/libros.csv',
        'positive-words.txt',
        'negative-words.txt'
    ]

    # Comprobar si los archivos existen, si no, crear y cargar según corresponda
    if not all(os.path.exists(file) for file in required_files):
        # Crear el directorio 'database' si no existe
        if not os.path.exists('database'):
            os.makedirs('database')

        # Si el archivo de libros no existe, cargarlo
        if not os.path.exists('database/libros.csv'):
            load_books()

    # Cargar las bases de datos necesarias
    load_database()
```

```
# Entrenar el modelo de sentimientos
model_sents, positive_words, negative_words = model_sentiments(developer_mode)

# Cargar el dataframe de la base de datos combinada
dataframe_bbdd = load_datasets_union()

# Entrenar el modelo de recomendaciones
recommendation_model = train_model(dataframe_bbdd, developer_mode)

# Ciclo principal de interacción con el usuario
while True:
    # Obtener la entrada del usuario (sentimiento y recomendación)
    prompt_sentimiento, prompt_recomendacion = input_user()

    # Salir si el usuario escribe 'chau'
    if prompt_sentimiento == "exit":
        break

    # Predecir sentimiento
    sentimiento = predict_sentiments(model_sents, positive_words, negative_words,
    prompt_sentimiento, developer_mode)

    # Predecir la categoría para la recomendación
    recommended_category = recommend_item(prompt_recomendacion,
    recommendation_model)

    if developer_mode:
        # Imprimir información detallada de la depuración
        print("Prompt sentimiento:", prompt_sentimiento)
        print("Prompt recomendación:", prompt_recomendacion)
        print("Categoría recomendada:", recommended_category)

    # Obtener la recomendación final (título y enlace)
    recommended_title, recommended_link = predict_recomendation(prompt_recomendacion,
    recommended_category, recommendation_model, dataframe_bbdd)

    # Mostrar los resultados con efectos visuales
    print_slow(f"Entiendo que hoy estás {sentimiento}")
    print_slow(f"Mi recomendación es: {recommended_title} y el enlace es:
    {recommended_link} \n")
```

Función chatbot

El código define una función chatbot() que inicializa un chatbot llamado ChatLPJ (Libros Películas Juegos) y guía al usuario a través de una serie de interacciones.

Configuración Inicial:

- Establece un parámetro para el "modo desarrollador".
- Desactiva las advertencias de python.
- Pregunta al usuario si desea activar el modo desarrollador.

Presentación y Reglas: El chatbot se presenta y explica tres reglas clave para interactuar:

- No entiende oraciones negativas.
- Las respuestas deben estar en formato "(cómo te sientes), (qué buscas)".
- Advierte que las respuestas pueden no ser perfectas debido a limitaciones del diseño.

Preparación y Ejecución:

El chatbot se prepara para interactuar con el usuario y llama a la función main() para comenzar el ciclo de interacción. Al finalizar, el chatbot se despide del usuario con "Nos vemos 🙌".

```
def chatbot() -> None:
    """
    Función principal que interactúa con el usuario, configura el modo de desarrollador,
    muestra un mensaje de bienvenida con instrucciones, y luego llama a la función principal
    que gestiona la predicción de sentimientos y recomendaciones.
    """

    # Activar el modo desarrollador según la respuesta del usuario
    developer_mode = False
    warnings.filterwarnings("ignore") # Ignorar advertencias durante la ejecución

    # Preguntar al usuario si desea activar el modo desarrollador
    if input("¿Desea activar el modo desarrollador? (s/n): \n") == "s":
        developer_mode = True

    # Limpiar la salida de consola antes de mostrar los mensajes de bienvenida
    clear_output(wait=False)

    # Mostrar mensajes de bienvenida y reglas de interacción
    print("-----")
```

```
print_slow("Hola, mi nombre es ChatLPJ, antes de empezar necesito darte algunas reglas")
print_slow('1) No entiendo negativos, por ejemplo "No quiero" o "No me siento bien"')
print_slow('2) Necesito que tus respuestas siempre sean "(como te sentis),(que buscas)" para entenderte mejor!')
print_slow('3) A veces mis respuestas no son las mejores, te pido disculpas, mis creadores tuvieron algunos problemas cuando me diseñaron')
print_slow("Ahora te pido un minuto para prepararme")
print("-----")

# Llamar a la función principal que maneja la predicción y recomendaciones
main(developer_mode)

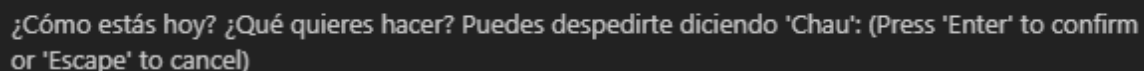
# Despedirse del usuario al finalizar
print("Nos vemos 🖐️")
```

Al ejecutar la sección "Chat", el algoritmo solicita primero un input para determinar si se activa o no el modo desarrollador. A continuación, ejecuta la función `main()`, que se encarga de validar y, si es necesario, cargar la base de datos, preentrenar los modelos y, finalmente, iniciar un bucle interactivo para interactuar con el usuario.

```
-----
Hola, mi nombre es ChatLPJ, antes de empezar necesito darte algunas reglas
1) No entiendo negativos, por ejemplo "No quiero" o "No me siento bien"
2) Necesito que tus respuestas siempre sean "(como te sentis),(que buscas)" para entenderte mejor!
3) A veces mis respuestas no son las mejores, te pido disculpas, mis creadores tuvieron algunos problemas cuando me diseñaron
Ahora te pido un minuto para prepararme
-----
```

Una vez terminado el proceso, dependiendo del entorno de ejecución tendremos dos alternativas.

- En caso de Visual Studio Code, tendremos una ventana emergente



```
¿Cómo estás hoy? ¿Qué quieres hacer? Puedes despedirte diciendo 'Chau': (Press 'Enter' to confirm or 'Escape' to cancel)
```

- En caso de Google Colab nos pondrá un cuadro de diálogo en el mismo entorno de ejecución.

```
-----
Hola, mi nombre es ChatLPJ, antes de empezar necesito darte algunas reglas
1) No entiendo negativos, por ejemplo "No quiero" o "No me siento bien"
2) Necesito que tus respuestas siempre sean "(como te sentis),(que buscas)" para entenderte mejor!
3) A veces mis respuestas no son las mejores, te pido disculpas, mis creadores tuvieron algunos problemas cuando me diseñaron
Ahora te pido un minuto para prepararme
-----
¿Cómo estás hoy? ¿Qué quieres hacer? Puedes despedirte diciendo 'Chau':

```

Al ingresar un prompt, el algoritmo ejecutará todo el proceso y nos escribirá de forma automática una respuesta.

Ejemplos de uso:

- 1) Prompt: *"Hoy fue un buen día, quiero ver una película de criminales intergalácticos"*

- ChatLPJ: *Entiendo que hoy estás feliz*

Mi recomendación es: Guardians of the Galaxy y el enlace es: nan

- 2) Prompt: *"Tenemos un día increíble con amigos, queremos un juego de mesa de aventuras y ciencia ficción"*

- ChatLPJ: *Entiendo que hoy estás feliz*

Mi recomendación es: Dungeons & Dragons: The Legend of Drizzt Board Game y el enlace es:

<https://boardgamegeek.com/boardgame/91872/dungeons-and-dragons-the-legend-of-drizzt-board-ga>

- 3) Prompt: *"Hoy fue un día horrible, quiero leer un libro de biología"*

- ChatLPJ: *Entiendo que hoy estás triste*

Mi recomendación es: On the Origin of Species y el enlace es: <https://www.gutenberg.org/ebooks/21153>

- 4) Prompt: *"Hoy me encuentro mal, quiero ver una película de comedia"*

- ChatLPJ: *Entiendo que hoy estás triste*

Mi recomendación es: Disaster Movie y el enlace es: nan

- 5) Hoy discuti con un amigo, me gustaria algo divertido para estar solo

- ChatLPJ: *Entiendo que hoy estás triste*

Mi recomendación es: Total Recall y el enlace es: nan

Conclusiones:

En la mayoría de los casos, el algoritmo ha proporcionado respuestas coherentes y esperadas, mientras se respeten las reglas establecidas. Las recomendaciones generadas cumplen los requisitos establecidos y la predicción de sentimientos se ha alineado adecuadamente con las entradas proporcionadas.

En el entrenamiento del modelo de sentimientos, obtuvimos métricas sobresalientes, especialmente al considerar que el modelo se basa en el análisis de palabras clasificadas como positivas o negativas, sin tener en cuenta la estructura sintáctica completa de las frases. A pesar de esta limitación, el rendimiento fue notable, demostrando la efectividad del enfoque basado en el análisis léxico.

```
Accuracy: 0.9996318114874816
Confusion Matrix:
[[1879   0]
 [   1 836]]
Classification Report:
              precision    recall  f1-score   support

     0           1.00        1.00        1.00       1879
     1           1.00        1.00        1.00        837

 accuracy          1.00          1.00          1.00       2716
 macro avg          1.00          1.00          1.00       2716
weighted avg          1.00          1.00          1.00       2716
```

Por otro lado, en el entrenamiento del modelo de recomendaciones, logramos resultados igualmente positivos mediante el uso de validación cruzada. Este enfoque permitió evaluar de manera robusta el desempeño del modelo, asegurando que las recomendaciones fueran consistentes y acertadas en distintos subconjuntos de datos.

```
Cross-validation scores: [0.99125 0.9925 0.99375 0.99375 0.995 ]
Mean cross-validation score: 0.99325
```

Sin embargo, cuando el prompt no es lo suficientemente explícito o no sigue las reglas definidas, el algoritmo puede experimentar confusión tanto en la predicción de sentimientos como en la recomendación. En estos casos, las respuestas pueden no ser precisas o no cumplir con las expectativas del usuario. Esto subraya la importancia de que los usuarios sigan las pautas especificadas para garantizar una interacción fluida y resultados precisos.

Por ejemplo:

Prompt: *“No quiero leer un libro pero si ver una pelicula, que no sea de ciencia ficcion sino de fantasia”*

- ChatLPJ: *Entiendo que hoy estas triste*

Mi recomendación es: Essays on the use and limit of the imagination in science y el enlace es: <https://www.gutenberg.org/ebooks/74654>

En este caso, el algoritmo utilizó la primera parte de la oración para determinar el sentimiento, excediendo sus capacidades para entender la intención completa del mensaje. Además, al enfrentar poca información precisa en la segunda parte de la oración, la recomendación generada no resulta coherente.

Prompt: *“No me siento bien, quiero un juego de mesa sobre musica”*

- ChatLPJ: *Entiendo que hoy estas feliz*

Mi recomendación es: HITSTER y el enlace es: <https://boardgamegeek.com/boardgame/318243/hitster>

En este caso, el algoritmo no capta la negatividad implícita en el enunciado, asociando “bien” como una palabra positiva y asumiendo un sentimiento alegre. No obstante, la recomendación es relevante.

Puntos de mejora:

Para optimizar el sistema, se plantea la implementación de un modelo de lenguaje más avanzado que comprenda mejor la estructura del lenguaje y pueda identificar matices complejos, como negaciones y sutiles variaciones en los mensajes. Se considera que los modelos basados en BERT u otros transformadores serían particularmente adecuados, ya que permiten una comprensión contextual profunda en ambas etapas de predicción, logrando respuestas más precisas y alineadas con la intención del usuario.

Otra posible mejora sería el desarrollo de una interfaz más dinámica que facilite una interacción fluida y adaptativa entre el usuario y el chatbot. Además, se consideró la incorporación de una funcionalidad de memoria que permita al algoritmo construir un perfil del usuario basado en patrones previos. Esto permitiría que palabras clave sean agregadas al prompt para mejorar las recomendaciones según intereses reiterados, como la preferencia por la ciencia ficción en varias búsquedas. Sin embargo, durante las pruebas se descubrió que no siempre se identificaba correctamente cuándo introducir estas palabras clave, lo que generaba recomendaciones que no siempre eran pertinentes.

Algo interesante para implementar, específicamente para el caso de los juegos de mesa, sería el uso de un algoritmo de árbol de decisiones que permita clasificar

según el número de jugadores, la edad, etc., para obtener mejores recomendaciones basadas en los detalles proporcionados en el prompt.

Por último, se propone utilizar la predicción de sentimientos para refinar aún más las recomendaciones según el estado emocional del usuario. Por ejemplo, si el usuario se siente triste, se evitaría sugerir una película de "romanticismo" u otros géneros que no se ajusten a su estado de ánimo. Aunque se intentó implementar esta funcionalidad, las predicciones de sentimientos no siempre generaron recomendaciones adecuadas, lo que dificultó alcanzar la precisión deseada en las recomendaciones.

Referencias

Sentiment Analysis Word Lists Dataset: [Sentiment Analysis Word Lists Dataset](#)