Programación Exploratoria 2023
(Ciencias Exactas - UNICEN)

Aprendizaje Basado en Proyectos

INFORME: ChatBot de **YouTube**

Alumna: Delfina Milagros Ferreri (delfiferreri23@gmail.com).
Docentes: Analía Amandi, Nelson Acosta.
Fecha de entrega: 27/11/23.

INTRODUCCIÓN

Este proyecto se desarrolla en el contexto de la materia Programación Exploratoria de la carrera Ingeniería de Sistemas (UNICEN) durante la cursada 2023. Su finalidad es la de aplicar los conocimientos adquiridos en la materia a través de la experimentación (aprendizaje basado en proyectos). Para llevarlo a cabo se ha utilizado el framework Rasa, una plataforma open-source de IA conversacional (documentación: https://rasa.com/docs/rasa/).

La consigna consiste en diseñar, implementar y entrenar a un agente conversacional orientado a la tarea de asistir a usuarios en la exploración de videos de YouTube. En líneas generales, el bot implementado es capaz de:

- ➔ Recordar el nombre del usuario si se da a conocer durante la sesión,
- ➔ Realizar búsquedas solicitadas por los usuarios a través de interacciones con la API de YouTube,
- ➔ Mantener un historial de dichas búsquedas,
- ➔ Repetir la última búsqueda del usuario - si éste lo pide - sin que este tenga que reingresar la clave de búsqueda,
- ➔ Mostrar el historial de búsqueda ante solicitud del usuario,
- ➔ Realizar recomendaciones tanto aleatorias como personalizadas a los usuarios aplicando un algoritmo de aprendizaje basado en redes neuronales,
- ➔ Recibir feedback sobre videos buscados/recomendados en la sesión,
- ➔ Mantener un historial de las recomendaciones junto con la valoración del usuario sobre la misma para aprender sobre las preferencias del usuario, sea éste conocido o no.

FUNCIÓN DE BÚSQUEDA

El usuario puede solicitar al bot que realice una búsqueda en YouTube, ingresando los términos de la búsqueda entre comillas dobles.

Reconocer la intención del usuario

Rasa presenta una herramienta muy útil para el reconocimiento del objetivo implícito en cada mensaje que los usuarios envían al agente: los *intents*. Estos actúan como etiquetas que clasifican las intenciones detrás de las interacciones de los usuarios, el sistema utiliza modelos NLU para clasificar el texto del usuario en estas categorías.

Para la función de búsqueda, en *domain.yml* se declaró como parte del dominio al *intent ask_for_search*. En *nlu.yml* se aportaron ejemplos para entrenar al agente de modo que pueda reconocer qué tipo de mensajes implican voluntad de búsqueda. Cuántos más ejemplos, mejor será el reconocimiento de dicha intención y menor la propensión a errores.

nlu.yml:

```
- intent: ask_for_search
  examples: |
    - Puedes buscar el video de "Enlightment Documentary"?
    - Reproduce la canción "Loca de Shakira"
    - Busca por favor "experiencias paranormales de usuarios de Reddit"
    - Estoy buscando ver algo relacionado con "psicología"
    - buscame un video de "datos curiosos que no conocías hace 5 minutos"
    - Reproducir "África by Toto"
    - Pon "los 10 mejores goles de messi"
    - buscar cualquier video de "Doc Tops"
    - Quiero buscar un video en específico sobre "la historia de la Tierra".
    - Estoy buscando un video en particular sobre "cómo hacer pan casero".
    - porfis buscar "tutorial de SQL"
```

```
? Your NLU model classified 'Quiero buscar "Elecciones Argentina 2023"'
with intent 'ask_for_search' and there are no entities, is this correct?
 (Y/n)
```

Acción de búsqueda de videos: interacción con la API de YouTube

En el archivo *stories.yml* se especifica que siempre que el usuario solicita una búsqueda, lo adecuado es inmediatamente ejecutar la acción *action_search_video* implementada en *actions.py*. La misma se encarga de extraer la clave de búsqueda y luego comunicarse con la API de YouTube (con la correspondiente *API key* que autoriza la interacción) mediante un request. Si la búsqueda arroja resultados, uno de ellos - al azar - se le entregará al usuario. Los resultados se traducen en la obtención de uno o más *video_id*, que son identificadores unívocos que YouTube asigna a cada video publicado en la plataforma. La extracción de la clave se puede realizar de dos maneras dependiendo del *intent*: si es una búsqueda simple extrae el texto entre comillas, mientras que si es una solicitud para repetir la búsqueda anterior la clave se extrae del *slot last_search_key*.

Una búsqueda exitosa provoca que se actualice el historial de búsquedas.
Si por alguna razón la búsqueda falla, se le notificará al usuario el motivo.

stories.yml:

```
- story: ideal search video with no fb
  steps:
  - intent: ask_for_search
  - action: action_search_video

- story: search video with repetition request
  steps:
  - intent: ask_for_search
  - action: action_search_video
  - intent: repeat_search
  - action: action_search_video

- story: search video with positive fb
  steps:
  - intent: ask_for_search
  - action: action_search_video
  - intent: positive_fb
  - action: utter_good
```

ejemplo

```
quiero buscar "Bill Gates Interview"

intent: ask_for_search 1.00

slot{"last_search_key": "Bill Gates Interview"}
slot{"search_history": [["Bill Gates Interview",
"f5NJQiY9AuY"]]}
action_search_video 0.96
Claro! Aquí te va el resultado de la búsqueda "Bill
Gates Interview" en YouTube:
https://www.youtube.com/watch?v=wmBj-5owOLA
slot{"last_search_key": "Bill Gates Interview"}
nt slots:
   username: Delfina, last_search_key: Bill Gates Interview,
', 'f5NJQiY9AuY']], recom_history: None, session_started_meta
```

actions.py: extracción de la clave de búsqueda e interacción con la API

```
class ActionSearchVideo(Action):
    # accion de búsqueda de videos en YouTube

    def name(self):
        return "action_search_video"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

        if (str(tracker.get_intent_of_latest_message()) == 'ask_for_search'):
            # extracción de la clave de búsqueda: texto entre comillas dobles
            search_key_url = re.findall(r'"([^"]*)"', tracker.latest_message['text'])[0]
        elif (str(tracker.get_intent_of_latest_message()) == 'repeat_search'):
            # extracción de la clave de búsqueda: la misma que la de la búsqueda anterior
            search_key_url = tracker.get_slot("last_search_key")
        else:
            return []

        if search_key_url != None:

            # clave de API - YouTube
            api_key = 'AIzaSyBOMEHk9dlGu6PQntgjBtydprgQO9JADJ8'

            # cantidad de resultados deseados
            num_resultados=7

            # generación del URL de la búsqueda de YouTube
            youtube_api_url = f'https://www.googleapis.com/youtube/v3/search?q={search_key_url}&key={api_key}&maxResults={n

            # búsqueda mediante solicitud a la API de YouTube
            response = requests.get(youtube_api_url)
            video_data = response.json()

            # si la búsqueda arroja resultados
            if 'items' in video_data:
                # arroja un link al azar de los n resultados hallados
                random_result = random.randint(0, num_resultados-1)
                video_id = video_data['items'][random_result]['id']['videoId']
                dispatcher.utter_message(text=f'Aquí te va el resultado de la búsqueda "{search_key_url}" en YouTube: '
                                         + f'https://www.youtube.com/watch?v={video_id}')
```

Con el *video_id* obtenido mediante la request a la API, se genera un link de tipo: *https://www.youtube.com/watch?v={video_id}* que redirecciona al usuario al video correspondiente en YouTube.

Historial de búsqueda

Cada vez que el usuario realiza una búsqueda exitosa, el bot registra esta búsqueda ingresando la misma al historial. Dicho historial se almacena en un *slot* de tipo lista llamado *search_history*, cuyos elementos obedecen el formato <*search_key, video_id*>. Este slot se actualiza automáticamente mediante una acción, al igual que el *slot last_search_key*.

Estos elementos ayudan a simular la memoria del bot. Gracias a esto en cualquier momento el usuario puede:

A. Pedirle al bot que le muestre el historial de búsquedas.
B. Pedirle al bot que repita la búsqueda anterior si el resultado arrojado no es lo que estaba buscando.

domain.yml:

```yaml
slots:
  search_history:
    type: list
    mappings:
      - type: custom
        action: action_update_search_history
```

actions.py:

```
""" B U S Q U E D A """

nueva_busqueda = None
> class ActionUpdateSearchHistory(Action): ...

> class ActionSetSlotLastSK(Action): ...

> class ActionSearchVideo(Action): ...

> class ActionDisplayHistory(Action): ...
```

actions.py:

- `(action_search_video)` - Al realizarse una búsqueda exitosa se guarda <*search_key* + *videoID*> en la variable global *nueva_busqueda*.

```python
# guarda la búsqueda para luego actualizar el historia
global nueva_busqueda
nueva_busqueda = (str(search_key_url), str(video_id))
# elimina lo que hay en search_key actualmente
return[SlotSet("search_key",None)]
```

- `(action_update_search_history)` - Inmediatamente después se ejecuta esta acción, que agrega el nuevo elemento (almacenado en *nueva_busqueda*) al historial de búsquedas.

```python
nueva_busqueda = None
class ActionUpdateSearchHistory(Action):
    # actualiza el historial de búsqueda de la conversación
    def name(self) -> Text:
        return "action_update_search_history"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any])

        global nueva_busqueda
        if nueva_busqueda != None:

            # obtengo el historial actual
            actual_history = tracker.get_slot("search_history") or []

            # agrego la nueva búsqueda a la lista
            actual_history.append(nueva_busqueda)

            nueva_busqueda = None

            # actualizo el slot de lista
            return [SlotSet("search_history",actual_history)]
```

A. **Ejemplo**: el usuario solicita ver el historial de búsqueda:

```
12                                                    quiero ver el historial
                                                    intent: ask_for_history 1.00

13   action_display_history 1.00
     H I S T O R I A L   D E   B U S Q U E D A S: [4 elemento/s]
     0. SK: Metaverso, Resultado:
     https://www.youtube.com/watch?v=WF0B84_QEbo
     1. SK: receta de
     pan casero, Resultado:
     https://www.youtube.com/watch?v=yCDTVzRQnD0
     2. SK: África de
     Toto, Resultado: https://www.youtube.com/watch?v=UQn-7GCh2r0
     3. SK: África de Toto, Resultado:
     https://www.youtube.com/watch?v=U1LB_OerHCE
```

B. La función de búsqueda de video está implementada de modo tal que arroja un resultado al azar de todos los que encuentra (max_resultados como máximo). Esto implica que al repetir la búsqueda con la misma clave, es probable que el resultado cambie. El *intent* *repeat_search* indica que el usuario desea repetir la búsqueda anterior y esto puede hacerse sin necesidad de reescribir la clave de búsqueda.

FUNCIÓN DE RECOMENDACIÓN

El usuario puede solicitar al bot que le recomiende un video de YouTube según sus preferencias personales. Para hacerlo se aplica un algoritmo de aprendizaje mediante redes neuronales.

Reconocer la intención del usuario

Al igual que en la funcionalidad anterior, es preciso reconocer la intención del usuario de solicitar una sugerencia. En este caso el *intent* se identifica por *ask_for_suggestion*. Como en el caso de la búsqueda, la acción asociada (*action_recommend_video*) presenta dos cursos de acción, esta vez dependiendo - no del *intent* - sino de los datos que se dispongan sobre la actividad de visualización del usuario. Estos datos provienen de dos fuentes: el archivo Prolog (*data.pl*) por un lado y el historial de recomendaciones de la sesión actual por el otro.

El rol de Prolog

El archivo *data.pl* funciona como una especie de base de datos del sistema. Hay una distinción entre la búsqueda y la recomendación en cuanto a amplitud, ya que la búsqueda de alguna manera tiene acceso a cualquier video de YouTube mientras que el bot solo puede recomendar videos que hayan sido previamente cargados al archivo Prolog.

En líneas generales *data.pl* se encarga de almacenar:

- **Usuarios** bajo hechos de tipo **`usuario(<nombre_usuario>).`**
- **Videos** bajo hechos de tipo **`video(<id>,<titulo>,<categoria>,<duracion>,<idioma>)`**
- **Categorias** bajo hechos de tipo **`categoria(<nombre_categoria>)`**
- **Opiniones** de usuarios sobre videos bajo hechos de tipo **`opinion(<nombre_usuario>,<id_video>,0/1)`** donde
  - **0** - indica que no le gustó el video;
  - **1** - indica que le gustó el video.

```
% USUARIOS: usuario(nombre_usuario)
usuario('Delfina').
opinion('Delfina','e9dZQelULDk',1).
opinion('Delfina','0UgiJPnwtQU',1).
opinion('Delfina','FavUpD_IjVY',1).
opinion('Delfina','kd2KEHvK-q8',1).
opinion('Delfina','G1p6rlDCxq0',0).
opinion('Delfina','B6Gi1uKr15Y',0).
```

```
% VIDEOS: video(videoID,Título,Categoria,Duracion,Idioma).
%
% videos de Animacion
video('e9dZQelULDk','Hapiness','Animacion','[0-5]','Any').
video('0UgiJPnwtQU','Cream by David Firth','Animacion','[5-15]','Eng').
video('FavUpD_IjVY','Cows & Cows & Cows','Animacion','[0-5]','Any').
video('xYnv8tHyOJE','Catopolis','Animacion','[10-15]','Eng').
%
% videos de Animales
video('eW6hheq8qfg','Guinea pigs go crazy for water melon','Animales','[6-15]','Any').
video('JQL25_hoQ1k','7 Spectacular Moths in Slow Motion!','Animales','[0-5]','Any').
video('2AP9dfBTcNQ','Cómo crecen los gatitos: de 0 día a 1 año','Animales','[41-60]',
%
% videos de Arte-Literatura
video('bvEHyeph1B4','Stanczyk or the Sad Clown Paradox','Arte y Literatura','[6-15]',
video('-qCngjk3nQw','The Most Disturbing Painting - A Different Take on Saturn Devour
```

Es sumamente importante conservar todos esos datos sobre los videos y las valoraciones de los usuarios sobre los mismos porque a partir de esa información es posible aplicar el algoritmo de recomendación basado en redes neuronales, de forma tal que puedan lograrse recomendaciones personalizadas, ajustadas a las preferencias de los usuarios en lugar de recomendaciones aleatorias.

Para que el chatbot pueda acceder y hacer uso pertinente de esta información, *data.pl* almacena también reglas. Entre las más importantes:

```
% devuelve una lista de todos los videos
all_videos(Videos):-
findall(video(ID,Titulo,Categoria,Duracion,Idioma),video(ID,Titulo,Categoria,Duracion,Idioma),Videos).

% devuelve los atributos relevantes de un video dado
get_video(ID,Video) :-
findall((Categoria,Duracion,Idioma),video(ID,_,Categoria,Duracion,Idioma),Video).

% devuelve una lista de todas las categorias
all_categorias(Categorias) :-
    findall(Nombre,categoria(Nombre),Categorias).

% devuelve una lista de todos los videos de cierta categoria
videos_por_categoria(Categoria,Videos) :-
    findall(Video,video_pertenece_a_categoria(Video,Categoria),Videos).

% devuelve una lista de todos los videos vistos por un usuario dado
videos_vistos_por_usuario(Usuario,Historial) :-
    findall((VideoID,Opinion),opinion(Usuario,VideoID,Opinion),Historial).

% devuelve una lista de todos los videos NO vistos por un usuario dado
videos_no_vistos_por_usuario(Usuario,NoVistos) :-
findall((VideoID,Categoria,Duracion,Idioma),(video(VideoID,_,Categoria,Duracion,Idioma),\+ opinion(Usuario,
VideoID, _)),NoVistos).
```

Es a partir de las reglas que es posible acceder a los datos desde *actions.py*. A continuación un ejemplo sencillo de una porción de código que consulta si el usuario es conocido (si ya se ha registrado en alguna sesión anterior) o si es un usuario nuevo:

```
from swiplserver import PrologMQI

# determino si el usuario es conocido - es decir - si existe en el archivo Prolog
if username:
    with PrologMQI(port=8000) as mqi:
        with mqi.create_thread() as prolog_thread:
            prolog_thread.query("consult('C:/Users/Delfina/OneDrive/Escritorio/Delfina/FA
            prolog_thread.query_async(f"usuario({username}).")
            conocido = prolog_thread.query_async_result()
            if conocido:
                prolog_thread.query_async(f"videos_vistos_por_usuario({username},H).")
                cant_vistos_pl = len(prolog_thread.query_async_result()[0]['H'])
```

actions.py - ActionRecommendVideo(Action) - run(...)

Acción de recomendación de videos: aleatoria y personalizada

Cómo se mencionó anteriormente, *action_recommend_video* cuenta con dos cursos de acción: recomendar al azar o recomendar según preferencias del usuario. La condición para que decante por una de las alternativas es sencilla: si no se tienen suficientes datos sobre el usuario, no tiene sentido aplicar un algoritmo de inteligencia artificial dado que la calidad de las recomendaciones depende directamente de la cantidad y la relevancia de los datos proporcionados por el usuario. En ausencia de información adecuada, la aplicación de un algoritmo de recomendación, siendo de por sí costosa, puede resultar poco efectiva e incumplir con las expectativas del usuario.

Por este motivo el bot toma la decisión de aplicar dicho algoritmo solo si la cantidad de datos - videos vistos junto con la opinión sobre los mismos - supera cierto número (por defecto fue seteado en 10, que en el contexto es bastante bajo). Si el registro es menor a este número, se aplica un algoritmo mucho más simple: recomendación aleatoria.

```
def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

    MIN_HISTORY = 10        # cantidad mínima de videos vistos por el usuario para poder recomendar con redes neuronales
    cant_vistos_sesion = 0  # cantidad de videos vistos segun sesion actual
    cant_vistos_pl = 0      # cantidad segun historial de videos vistos en archivo Prolog

    username = tracker.get_slot("username")

    # determino si el usuario es conocido - es decir - si existe en el archivo Prolog
    if username:···

    # cantidad de videos vistos por el usuario
    ACTUAL_HISTORY = cant_vistos_sesion + cant_vistos_pl
    print(ACTUAL_HISTORY)

    # si no se tiene registro suficiente del usuario: se recomienda un video al azar
    if ACTUAL_HISTORY < MIN_HISTORY:
        print('Registro insuficiente: la recomendación es al azar')
        recom = self.random_recom({username})
    # si se tiene registro suficiente: se ejecuta el algoritmo de redes neuronales
    else:
        print('Registro suficiente: la recomendacion es personalizada')
```

actions.py - ActionRecommendVideo(Action)

Algoritmo de recomendación aleatoria

```python
# devuelve un video (no visto por el usuario) al azar
def random_recom(self,username):

    with PrologMQI(port=8000) as mqi:
        with mqi.create_thread() as prolog_thread:
            prolog_thread.query("consult('C:/Users/Delfina/OneDrive/Escritorio/Delfina/FACULTAD/PExp/

            prolog_thread.query_async(f"cant_videos(C).")
            cant_videos = prolog_thread.query_async_result()[0]['C']    # cantidad total de videos

            prolog_thread.query_async(f"all_videos(L).")
            resultados = prolog_thread.query_async_result()             # lista de todos los videos

            # extraer los ID y filtrar los videos vistos por el usuario (si es conocido)
            candidatos = []
            i = 0
            while i<cant_videos:
                video_id = resultados[0]['L'][i]['args'][0]
                if username:
                    prolog_thread.query_async(f"ha_visto('{video_id}',{username}).")
                    visto = prolog_thread.query_async_result()
                    if not visto:
                        candidatos.append(f'{video_id}')
                i=i+1

            random_index = random.randint(0, len(candidatos)-1)        # index de video al azar
            recom = list(candidatos)[random_index]
    return recom
```

actions.py - ActionRecommendVideo(Action) - random_recom()

Algoritmo basado en redes neuronales

En ***neural_networks.py*** se encuentra escrita la función de recomendación personalizada. Para poder llamarla desde ***actions.py*** primero fue preciso importar el módulo de modo que la reconozca:

```python
from neural_networks import personalized_recom
```

El algoritmo lee un archivo JSON llamado *viewed_videos.json* que contiene información sobre todos los videos vistos del usuario. En el próximo punto se cubre de qué manera es cargado el mismo. Utiliza *LabelEncoder* para transformar las columnas 'Categoria', 'Duracion' e 'Idioma' del dataset a valores numéricos. Luego define un modelo de red neuronal simple con una capa de entrada y dos capas ocultas y lo compila. Después de entrenar el modelo, lo evalúa y muestra la pérdida y la precisión.

Para dar con una buena recomendación consulta a *data.pl* por los videos no vistos por el usuario y contempla a todos ellos como candidatos a recomendación. Itera sobre ellos hasta encontrar alguno que podría gustarle, intentando predecir la opinión del usuario sobre cada candidato utilizando el modelo de red neuronal previamente entrenado.

Si la predicción supera un umbral (0.5), el bot considera que es una buena recomendación y devuelve el ID del video recomendado para que *action_recommend_video* se lo presente al usuario mediante el siguiente mensaje:

```python
    # con el dataset cargado ejecuto neural_networks para todos los videos no vistos por el usuario
    recom = personalized_recom(data)

    # vaciar viewed_videos
    data.clear()

dispatcher.utter_message(f"Por supuesto! Tal vez esto te guste... https://www.youtube.com/watch?v={recom}")
```

código completo de ***neural_networks.py***

neural_networks.py

```python
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from swiplserver import PrologMQI

# algoritmo de recomendacion personalizada - retorna id_video
def personalized_recom(username):

    DATA_PATH = 'C:/Users/Delfina/OneDrive/Escritorio/Delfina/FACULTAD/PExp/YT_ChatBot/actions/data.pl'

    # codificación previa: se hace antes de crear el dataset a partir de los datos en Prolog
    # esto es para evitar errores de tipo 'unknown label' en LabelEncoder
    label_encoder_cat = LabelEncoder()
    label_encoder_dur = LabelEncoder()
    label_encoder_lang = LabelEncoder()
    with PrologMQI(port=8000) as mqi:
      with mqi.create_thread() as prolog_thread:
        prolog_thread.query(f"consult('{DATA_PATH}')")
        # obtengo lista de categorias para encodear
        prolog_thread.query_async(f"all_categorias(V)")
        categorias = prolog_thread.query_async_result()[0]['V']
        print(np.array(categorias))
        prolog_thread.query_async(f"all_duraciones(V)")
        duraciones = prolog_thread.query_async_result()[0]['V']
        prolog_thread.query_async(f"all_idiomas(V)")
        idiomas = prolog_thread.query_async_result()[0]['V']

        label_encoder_cat.fit(np.array(categorias))
        label_encoder_dur.fit(np.array(duraciones))
        label_encoder_lang.fit(np.array(idiomas))
    # creacion del dataset
    df = pd.read_json('viewed_videos.json')
    print(df)

    # codificacion de los atributos discretos del dataset
    df['Categoria']=label_encoder_cat.transform(df['Categoria'])
    df['Duracion']=label_encoder_dur.transform(df['Duracion'])
    df['Idioma']=label_encoder_lang.transform(df['Idioma'])

    # definicion de los datos de entrenamiento
    x = df[['Categoria','Duracion','Idioma']].values
    y = df['Opinion'].values
    x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2,random_state=42)

    # creacion del modelo
    model = keras.Sequential([
        keras.layers.Input(shape=(x.shape[1],)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(32, activation='relu'),
        keras.layers.Dense(1, activation='sigmoid')
    ])

    # entrenamiento del modelo
    model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
    model.fit(x_train,y_train,epochs=10,batch_size=64,validation_data=(x_test,y_test))
    loss, accuracy = model.evaluate(x_test,y_test)
    print(f'Loss: {loss}, Accuracy: {accuracy}')
```

```python
# definición de los videos candidatos: tuplas de prueba para el algoritmo
# los candidatos a recomendación son los videos no vistos del ususario
if username:
  with PrologMQI(port=8000) as mqi:
    with mqi.create_thread() as prolog_thread:
        prolog_thread.query(f"consult('{DATA_PATH}')")
        prolog_thread.query_async(f"videos_no_vistos_por_usuario({username},V)")
        candidatos = prolog_thread.query_async_result()[0]['V']

# itera sobre los candidatos para hallar una recomendacion
i=0
while i<len(candidatos):

  # extracción de los datos del video i
  recom = candidatos[i]['args'][0]
  cat = candidatos[i]['args'][1]['args'][0]
  dur = candidatos[i]['args'][1]['args'][1]['args'][0]
  idioma = candidatos[i]['args'][1]['args'][1]['args'][1]
  # codificacion de los atributos discretos
  sample_video = np.array([[label_encoder_cat.transform([f'{cat}']),
                            label_encoder_dur.transform([f'{dur}']),
                            label_encoder_lang.transform([f'{idioma}'])]])

  # utiliza el modelo para predecir si el video puede gustar o no
  prediction = model.predict(sample_video)
  if prediction >= 0.5:
    print(f'Recomendación hallada! https://www.youtube.com/watch?v={recom}')
    return recom
  i=i+1

return None
```

***neural_networks.py** en ejecución*

```
PS C:\Users\Delfina\OneDrive\Escritorio\Delfina\FACULTAD\PExp\YT_ChatBot\actions> python neural_networks.py
       VideoID            Categoria  Duracion Idioma  Opinion
0   e9dZQelULDk            Animacion     [0-5]    Any        1
1   0UgiJPnwtQU            Animacion    [5-15]    Eng        1
2   FavUpD_IjVY            Animacion     [0-5]    Any        1
3   kd2KEHvK-q8    Misterio y Terror   [16-25]    Eng        1
4   G1p6rlDCxq0  Historia y Politica  [61-180]    Eng        0
5   B6Gi1uKr15Y    Misterio y Terror   [26-40]    Esp        0
6   e9dZQelULDk            Animacion     [0-5]    Any        1
7   0UgiJPnwtQU            Animacion    [5-15]    Eng        1
8   FavUpD_IjVY            Animacion     [0-5]    Any        1
9   kd2KEHvK-q8    Misterio y Terror   [16-25]    Eng        1
10  G1p6rlDCxq0  Historia y Politica  [61-180]    Eng        0
11  B6Gi1uKr15Y    Misterio y Terror   [26-40]    Esp        0
2023-11-27 22:45:33.322025: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is opti
 oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical oper
VX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/10
```

...

```
Epoch 9/10
1/1 [==============================] - 0s 33ms/step - loss: 0.5659 - accuracy: 0.8889 - val_loss: 0.7279 - val_accuracy:
0.6667
Epoch 10/10
1/1 [==============================] - 0s 32ms/step - loss: 0.5562 - accuracy: 0.8889 - val_loss: 0.7229 - val_accuracy:
0.6667
1/1 [==============================] - 0s 24ms/step - loss: 0.7229 - accuracy: 0.6667
Loss: 0.7229215502738953, Accuracy: 0.6666666865348816
1/1 [==============================] - 0s 77ms/step
Recomendación hallada! https://www.youtube.com/watch?v=xYnv8tHyOJE
```

Uso de JSON para manipulación de datasets: comunicación entre archivos.

JavaScript Object Notation (JSON) es un formato de intercambio de datos ligero y fácil de leer que se utiliza comúnmente para transmitir datos entre diferentes partes de una aplicación.

Al querer implementar el algoritmo de *neural_netwoks.py* surgió el desafío de tener que "transportar" la información desde Prolog y el historial de recomendaciones a un dataset perfectamente estructurado que me permita ejecutar el algoritmo sin errores. La mejor solución fue implementar una clase en *actions.py* cuya finalidad fuera volcar los datos en un archivo de tipo JSON (*viewed_videos.json*). Con una clase ***JSONDataset*** de tres métodos sencillos pudo resolverse el problema de la comunicación entre ambos archivos '*.py'*. Los métodos son (1) carga/load del archivo, (2) agregar elemento/fila al dataset y (3) vaciar archivo - dejarlo libre para la próxima ejecución -.

```python
""" R E C O M E N D A C I O N """

class JSONDataset():

    # metodo constructor
    def __init__(self, file_path) -> None:
        self.json_path = file_path

    # insertar un nuevo dato/fila
    def add(self, element):
        data = self.load()
        data.append(element)
        with open(self.json_path, 'w') as file:
            json.dump(data, file, indent=2)

    # cargar el archivo
    def load(self):
        if os.path.isfile(self.json_path):
            with open(self.json_path,"r") as arch:
                archivo=json.load(arch)
                arch.close()
        else:
            archivo={}
        return archivo

    # vaciar el archivo
    def clear(self):
        with open(self.json_path, 'w') as arch:
            arch.write("[]")
            arch.close()
```

Formato de viewed_videos.json:

```json
[
  {
    "VideoID": "e9dZQelULDk",
    "Categoria": "Animacion",
    "Duracion": "[0-5]",
    "Idioma": "Any",
    "Opinion": "1"
  },
  {
    "VideoID": "0UgiJPnwtQU",
    "Categoria": "Animacion",
    "Duracion": "[5-15]",
    "Idioma": "Eng",
    "Opinion": "1"
  },
  {
    "VideoID": "FavUpD_IjVY",
    "Categoria": "Animacion",
    "Duracion": "[0-5]",
    "Idioma": "Any",
    "Opinion": "1"
  },
  {
    "VideoID": "kd2KEHvK-q8",
    "Categoria": "Misterio",
    "Duracion": "[16-25]",
    "Idioma": "Eng",
    "Opinion": "1"
  }
]
```

Clase JSONDataset

dataset - viewed_videos.json:

[{"VideoID": "e9dZQelULDk","Categoria": "Animacion","Duracion": "[0-5]","Idioma": "Any","Opinion": "1"},
{"VideoID": "0UgiJPnwtQU","Categoria": "Animacion","Duracion": "[5-15]","Idioma": "Eng","Opinion": "1"},
{"VideoID": "FavUpD_IjVY","Categoria": "Animacion","Duracion": "[0-5]","Idioma": "Any","Opinion": "1"},
{"VideoID": "kd2KEHvK-q8","Categoria": "Misterio y Terror","Duracion": "[16-25]","Idioma": "Eng","Opinion": "1"},
{"VideoID": "G1p6rlDCxq0","Categoria": "Historia y Politica","Duracion": "[61-180]","Idioma": "Eng","Opinion": "0"},
{"VideoID": "B6Gi1uKr15Y","Categoria": "Misterio y Terror","Duracion": "[26-40]","Idioma": "Esp","Opinion": "0"}]

En la siguiente porción de código se muestra cómo se procede ante las recomendaciones personalizadas. En resúmen los pasos son:

1. Se crea una instancia de la clase JSONDataset para referenciar y manipular el contenido de viewed_videos.json.
2. Si el usuario es conocido es porque existe en Prolog, en cuyo caso se carga el dataset con los datos de visualización asociados al mismo en data.pl.
3. Sea conocido o no, se cargan en el dataset los datos de visualización de la sesión actual. Estos datos se extraen del historial de recomendación almacenado en uno de los slots.
4. Ambos métodos de carga del dataset están implementados dentro de la clase ActionRecommendVideo(), disponibles en el código fuente.

```
# si se tiene registro suficiente: se ejecuta el algoritmo de redes neuronales
else:

    print('Registro suficiente: la recomendacion es personalizada')
    # creo el dataset
    data = JSONDataset("ChatBot_new\\actions\\viewed_videos.json")

    # cargo el dataset
    if conocido:
        self.cargar_dataset_pl(data,username)
    # agrego los datos del historial de recomendaciones
    self.cargar_dataset_sesion(data)

    # con el dataset cargado ejecuto neural_networks para todos los videos no vistos por el usuario
    recom = personalized_recom(data)

    # vaciar viewed_videos
    data.clear()
```

La importancia del feedback en la recomendación

Al igual que en el caso de la búsqueda, existe un historial de recomendación. Este no está disponible para el usuario, sino que existe solo para recopilar información a usarse en el algoritmo de recomendación.

Es importante recalcar que, ante una sugerencia de video del bot al usuario, si el usuario no da un feedback inmediatamente (negativo o positivo, detectado por intents *positive_fb* y *negative_fb* respectivamente) la recomendación no será agregada al historial de porque no sería de valor para el algoritmo de recomendación - ¿cómo utilizar el dato si no se sabe si al usuario le gustó o no? -.

```
1   action_listen
2                                                    Hola recomendar
                                                intent: ask_for_suggestion 1.00
3   action_recommend_video 1.00
    Por supuesto! Tal vez esto te guste...
    https://www.youtube.com/watch?v=S1g1cQjUguI
    action_listen 1.00
4                                                    excelente video!
                                                intent: positive_fb 1.00
5   slot{"recom_history": [["S1g1cQjUguI", 1]]}
    utter_good 1.00
    Sabía que te gustaría! Hay más de donde vino ese...
```

COMENTARIOS, PROBLEMAS y POSIBLES OPTIMIZACIONES

1. El algoritmo de *neural_networks* funciona, pero no es muy efectivo debido a que la cantidad de datos es demasiado pequeña.
2. Se necesita una base de datos más grande e incluso expansible - ¿Hay manera de que los videos buscados por los usuarios puedan ir incorporándose a la base representada en Prolog? -.
3. La búsqueda podría enriquecerse con filtros y cuestiones a flexibilizar. Conocer más cómo funciona en profundidad la API de YouTube traería ventajas de este tipo. Por ejemplo en cuanto a la categorización de videos: podrían aprovecharse las categorías planteadas por YouTube en vez de crearlas manualmente.
4. Podrían recomendarse y buscarse canales de YouTube en vez de solo videos.
5. Otra opción sería permitirle al usuario interactuar más con YouTube: dejar likes, comentar, guardar en “Ver más tarde”. Esta funcionalidad aportaría comodidad para los usuarios.
6. A nivel chatbot conversacional: podría mejorar mucho en las interacciones. Presenta varios errores al conversar, a veces malinterpreta los *intents* pero es cuestión de refinarlo y volver a entrenar.