PROYECTO SISTEMA DE RECOMENDACIONES CRUZADAS

GERARDO BLÁZQUEZ MORENO

Índice

1. Introducción

Este proyecto implementa un sistema de **recomendaciones cruzadas** capaz de relacionar películas, series, libros y videojuegos. Se basa en un enfoque **cliente-servidor**, donde:

- El **backend en Python** gestiona la lógica de recomendación.
- El **frontend en Astro + Tailwind + JS** muestra los resultados dentro de un portfolio web.

Consiguiendo encontrar recomendaciones según los gustos del usuario y mostrárselos de manera visual y clara.

2. Historia y evolución

2.1. Primeros pasos: Dataset unificado

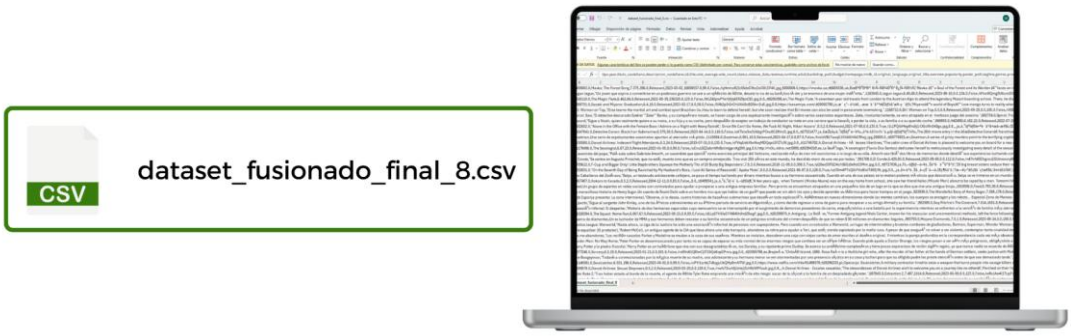
El sistema comenzó con la **unificación de datasets de Kaggle**:

- Se **fusionaron datos** de **películas** (**6000**), **series** (**7000**), **libros** (**2000**), y **videojuegos** (**14000**), en un solo dataset (***dataset_fusionado_final_8.csv***).
- Se agregó una columna **tipo** para diferenciar **cada categoría**.

Este dataset contenía información básica (títulos, puntuaciones, fechas, géneros).

Sin embargo, los datos estaban **incompletos** y **desbalanceados**:

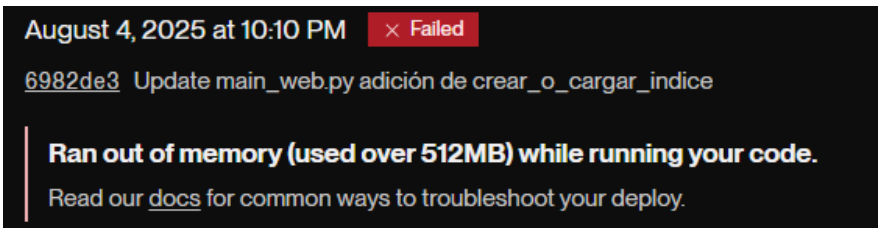
- Muchos títulos estaban en inglés.
- Faltaban sinopsis, autores, directores o imágenes.
- Había registros inconsistentes o duplicados.



2.2. Primer enfoque: embeddings con Sentence-Transformers

En la primera versión, el sistema se construyó usando **Sentence-Transformers** para generar **embeddings** de **cada obra** (película, libro, serie, videojuego) y un índice **FAISS** para realizar búsquedas semánticas.

- **Ventaja**: gran capacidad de capturar similitud contextual y semántica.
- **Problema**: al desplegar en **Render** (**versión gratuita**), el consumo de **memoria RAM superaba los límites**, lo que hacía **inviable** esta aproximación en producción.



2.3. Optimización: modelo TF-IDF + similitud de coseno

Para resolver el problema de infraestructura, se **rediseñó el motor usando un enfoque más ligero y eficiente**:

- **TF-IDF**: vectorización basada en frecuencia de palabras (***overview, keywords, títulos***).
- **Similitud de coseno**: comparación entre vectores TF-IDF para encontrar obras similares.
- **Filtrado por géneros y reglas**: asegura que las recomendaciones tengan coherencia temática.

Este cambio permitió mantener **buen rendimiento en calidad** de recomendaciones, pero **reduciendo drásticamente el uso de memoria** y **tiempo** de cómputo.

2.4. Enriquecimiento con constantes.py

Para mejorar la calidad de los resultados, se creó un archivo **constantes.py** con **diccionarios** de títulos, pudiendo utilizar la **clave externa** y conocimiento adicional:

- **Sinónimos de géneros** y **palabras clave** → *Ej: "sci-fi" = "ciencia ficción"*.
- **Sagas** y **franquicias** → *detectar colecciones como Marvel, Pixar, Anime, etc.*
- **Combinaciones prohibidas** de géneros → *evitar mezclas sin sentido (ej: "documental + comedia romántica")*.
- **Géneros forzados** → *corregir casos en que el API devuelve datos incompletos*.
- **Traducciones** de **títulos** → *homogeneizar búsquedas en inglés y español. Así luego poder utilizar la traducción para **emparejarlo** con la **clave externa** de los **diccionarios** y **enriquecer** con el **contenido de estos diccionarios***.

2.5. Sistema de *Scoring* personalizado

Además de la similitud semántica (**TF-IDF + coseno**), se incorporó un sistema de **scoring multicriterio** para **refinar** las recomendaciones.

Factores considerados:

1. Similitud por **overview** (vectorización).
2. **Fuzzy Matching** en títulos (corrección de errores, similitud de nombres).

3. **Coincidencia de géneros**:
    - Géneros **fuertes** (ej: animación, ciencia ficción, fantasía) → +**1.0**.
    - Géneros **débiles** (ej: drama, comedia, romance) → +**0.3**.
4. **Primer género coincidiente** entre origen y destino → +**0.8**.
5. **Productora/estudio** detectado (*Pixar*, *Disney*, *Ghibli*, *Marvel*, *DC*...) → **bonus** entre +**0.4** y +**0.8**.
6. **Keywords** especiales en **overview** o **metadatos** → hasta +**0.5**.
7. **Recencia**: bonus si el año es mayor a un límite definido (> **1990**).
8. **Fallback bonus**: si un título es **muy similar pero no tiene keywords** o **géneros** → +**0.5**.

Extracto de código

```python
# --- SCORING ---
def score_orden(c):
    generos_c = c.get("generos", []) or []
    primer_genero_c = generos_c[0] if generos_c else None
    primer_genero_obj = genero_obj_map[0] if genero_obj_map else None
    coincide_primer_genero = primer_genero_c == primer_genero_obj

    estudio = " ".join(
        (c.get("production_companies") or c.get("productora") or "")
        if isinstance(c.get("production_companies") or c.get("productora") or "", list)
        else [str(c.get("production_companies") or c.get("productora") or "")]
    )
    estudio = str(estudio).lower()
    BONUS_ESTUDIOS = {
        "pixar": 0.8, "disney": 0.7, "ghibli": 0.6, "dreamworks": 0.5,
        "illumination": 0.5, "marvel": 0.7, "dc": 0.6, "mapa": 0.4
    }
    bonus_estudio = 0.0
    estudio_detectado = None
    for clave, bonus in BONUS_ESTUDIOS.items():
        if clave in estudio:
            bonus_estudio = bonus
            estudio_detectado = clave
            break

    generos_fuertes = {"animacion", "animation", "aventura", "adventure", "familia", "family",
                       "fantasia", "fantasy", "science fiction", "musical", "ciencia ficcion", "ciencia ficción",
                       "superhero", "accion", "action", "marvel", "pixar", "disney"}
    generos_debiles = {"drama", "comedia", "romance"}
    bonus_genero = 0.0
    detalle_generos = []
    for g in generos_c:
        g_norm = unidecode(g).lower()
        if g_norm in generos_fuertes:
            bonus_genero += 1.0
            detalle_generos.append(f"{g}(+1.00)")
        elif g_norm in generos_debiles:
            bonus_genero += 0.3
            detalle_generos.append(f"{g}(+0.30)")
        else:
            detalle_generos.append(f"{g}(0.00)")

    peso_fuzz = 0.05 if query_corta else 0.1
    similitud_val = max(c.get("similitud", 1) * 0.25, 0.3)
    fuzz_val = fuzz.token_set_ratio(str(query or ""), str(c.get("titulo", ""))) / 100 * peso_fuzz
    primer_genero_val = (0.8 if coincide_primer_genero else 0)
    keywords_val = min(
        contiene_palabra_clave_especial(c.get("keywords", "")) +
        contiene_palabra_clave_especial(c.get("overview", "")),
        0.5
    )
    tipo_val = tipo_pesos.get(c.get("tipo"), 1.0) * 0.05
    anio_val = (0.15 if (obtener_anio(c) and obtener_anio(c) > antiguedad_limite) else 0)
    titulo_simil = fuzz.token_set_ratio(str(query or "").lower(), str(c.get("titulo", "") or "").lower()) / 100
    fallback_title_bonus = 0.0
    if keywords_val == 0 and bonus_genero == 0 and titulo_simil > 0.8:
        fallback_title_bonus = 0.5

    generos_obj_set = set(genero_obj_map or [])
    generos_c_set = set(generos_c)
    if "anime" in generos_obj_set and "anime" in generos_c_set:
        bonus_genero += 0.7

    score = (
        similitud_val + fuzz_val + bonus_genero + primer_genero_val +
        keywords_val + tipo_val + anio_val + bonus_estudio + fallback_title_bonus
    )
```

2.6. Enriquecimiento con APIs externas

Para mejorar la calidad del dataset se integraron varias **APIs externas**:

Esto permitió tener datos **más completos y coherentes**.
El sistema también contempló las **limitaciones de llamadas y bloqueos** de las APIs:

- Se implementó un sistema de **caché en disco** para no repetir llamadas.
- Se **normalizaron géneros** y **keywords** para unificar criterios.

2.7. De sistema local a despliegue en Render

Inicialmente todo el código era **monolítico** (un solo archivo main.py con toda la lógica).
Funcionaba en local, pero al intentar desplegarlo en **Render** fue necesario:

- **Modularizar** el código.
- **Separar responsabilidades** en diferentes archivos.
- Crear un entorno más robusto con soporte para **Docker, Railway y Gunicorn**.

2.8. **Estructura final del proyecto**

Tras la **modularización**, el proyecto quedó dividido en **varios archivos** y **módulos**, cada uno con una función específica:

Archivos principales de infraestructura

- **.dockerignore / Dockerfile** → configuración de Docker para empaquetar y desplegar el servicio.
- **render.yaml / Procfile / gunicorn.conf.py / .railwayignore** → archivos de despliegue en Render y Railway, con configuración de servidor WSGI (Gunicorn).
- **.env.example** → plantilla de variables de entorno (API keys, configuraciones).
- **requirements.txt / requeriments-dev.txt** → dependencias del proyecto (entorno de producción y de desarrollo).

2.9. **Backend modular**

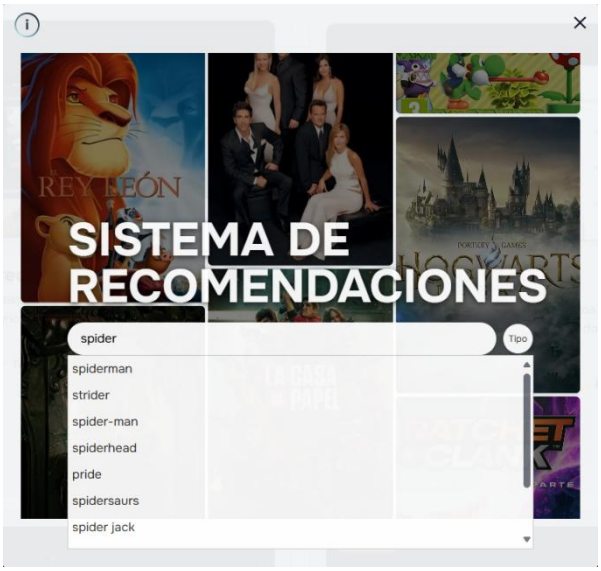
- **main_web.py** → es la **entrada principal del servidor Flask**, pensado específicamente para el entorno de **producción** de un **sistema de recomendaciones multimedia**.

  Se encarga de **inicializar el servidor**, habilitar **CORS** para manejo seguro de peticiones externas y cargar el conjunto de datos ya procesado desde **index.py** junto a módulos dedicados para lógica de negocio (**caché, recomendador, constantes**).

    - La arquitectura garantiza **respuestas rápidas y enriquecidas**, accediendo a la **caché** y solo llamando a **APIs externas** cuando faltan datos, y está preparada para **despliegue real** (por ejemplo, con **Gunicorn** o **Uvicorn**), asegurando **alta disponibilidad y escalabilidad** para frontend o integraciones externas.

Define **endpoints HTTP** para **salud del sistema**, **registro de eventos** y los dos pilares funcionales:

- **/autocomplete** → responde con **sugerencias instantáneas** usando **fuzzy match** en los títulos disponibles.



- **/recomendar** → recibe los parámetros de consulta del usuario (**título, tipo, género, exclusiones…**), busca la información básica, invoca la función central para obtener recomendaciones, completa y filtra los resultados y asegura que cada uno tenga **metadatos clave** y **trailers en castellano** (Del título buscado para que el usuario sepa sobre qué título se ha hecho la búsqueda).

- **index.py** → es el **otro punto de entrada** y se utiliza principalmente para **pruebas locales**, **validaciones** o **variantes de ejecución**; su propósito es **preparar, limpiar y procesar el dataset multimodal** (**películas, series, libros, videojuegos**), descargándolo si es necesario y normalizando campos como **tipo, géneros, títulos y descripciones**.

  Realiza una **limpieza exhaustiva**, homogeneiza todo el conjunto de datos y rellena los campos necesarios para garantizar la **consistencia**.
  Al final del proceso, inicializa la **vectorización avanzada mediante TF-IDF** sobre la columna **'overview'**, eliminando **stopwords** y dejando el **DataFrame, el vectorizador y la matriz listos en memoria RAM** para búsquedas y rankings eficientes.

  Toda esta lógica queda **expuesta globalmente**, permitiendo que el **servidor web** o cualquier otro proceso acceda siempre a los **datos preparados**. Así, **index.py** puede lanzarse por separado para **pruebas, sustituciones de datasets, reindexados**, o **integraciones con nuevas fuentes** sin afectar la **estabilidad** o el **rendimiento** del sistema principal usado en **producción**.

Módulos principales

- **recomendador.py** → motor central de recomendaciones:
  normaliza consultas, aplica reglas, busca en APIs, filtra y puntúa.
- **cache.py** → gestiona caché en disco y llamadas a APIs externas.
- **scoring.py** → funciones de puntuación y ranking de candidatos.
- **validacion.py** → validaciones extra para asegurar consistencia de datos.
- **utils.py** → funciones auxiliares reutilizables (normalización, helpers).
- **constantes.py** → reglas fijas (géneros forzados, combinaciones prohibidas, palabras clave especiales)

  .

2.10. Datos y modelos

- **dataset_fusionado_final_8.csv** → dataset base unificado.
- **pelicula.index, serie.index, libro.index, videojuego.index** → índices FAISS para búsquedas vectoriales (embeddings). No usado en el modelo final

- **pelicula_metadata.pkl, serie_metadata.pkl, libro_metadata.pkl, videojuego_metadata.pkl** → metadatos asociados a cada índice (títulos, IDs, etc.).

2.11. Resultado final

Gracias a esta **evolución**:

- El sistema pasó de un enfoque **pesado** y **difícil** de desplegar a uno **optimizado** y **escalable**.
- El motor ahora combina **TF-IDF** + **similitud de coseno** + **reglas de constantes.py** + **scoring multicriterio**.
- Esto garantiza recomendaciones:
    - **Más rápidas.**
    - **Coherentes** en **géneros** y **franquicias**.
    - **Adaptadas** a la **infraestructura gratuita de Render**.

3. Arquitectura General

Cliente (Frontend)

- Construido en **Astro** con estilos en **TailwindCSS** y lógica en **JavaScript**.
- Consume los **endpoints REST** del **backend**.
- Muestra los resultados con **animaciones**, **scroll** suave y **UI responsive**.

Servidor (Backend en Python)

- **recomendador.py**: motor principal de recomendaciones.
    - **Normalización** de títulos y **detección** de **géneros**.
    - **Aplicación de reglas** de **constantes.py** (palabras_clave_especial, géneros forzados, prohibidos, sagas).
    - **Enriquecimiento** de datos mediante **APIs externas**.
- **cache.py**: sistema de caché para reducir latencia y llamadas innecesarias a las APIs.

4. Motor de búsqueda (recomendador.py)

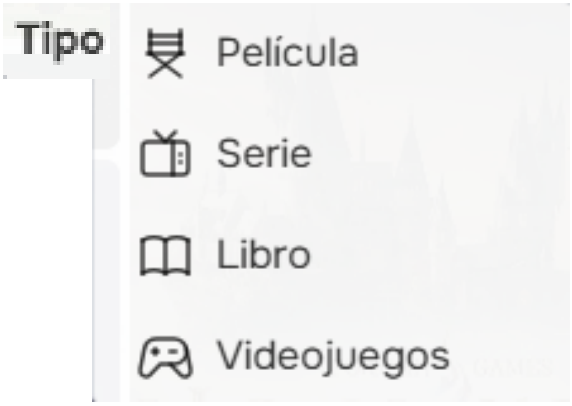
Explicación de recomendador.py

Este archivo es el **motor principal del sistema de recomendaciones**. Aquí se define la lógica para buscar, filtrar, enriquecer y devolver recomendaciones relevantes.

Funciones y responsabilidades principales

4.1. Normalización de entradas

- ***normalizar_titulo*** (**titulo**) → convierte títulos a **minúsculas**, **sin acentos**, con **snake_case**.
  Ejemplo: "*Spider-Man: Homecoming*" → "*spider_man_homecoming*".
- ***traducir_titulo_usuario***(**titulo**) → compara el título ingresado con un diccionario (**TRADUCCION_TITULOS**) y, si no hay coincidencia exacta, usa **fuzzy matching** (**rapidfuzz**) **para corregir errores de escritura**.
- ***normalizar_tipos_destino***(lista_tipos) → asegura que el tipo de destino sea válido (**pelicula**, **serie**, **libro**, **videojuego**).



4.2. Manejo de duplicados

- ***eliminar_duplicados***(recs, titulo_origen)
  Filtra resultados repetidos y evita devolver la misma obra como recomendación.
  - Si el título es **exactamente** el **mismo**, lo **elimina**.
  - **Excepción**: permite **secuelas** (ej: *Avengers* → *Avengers: Endgame*).

4.3. Procesamiento de keywords

- ***extraer_keywords_desde_overview***(overview)
  Si una **API** no devuelve **keywords**, se **extraen palabras relevantes del resumen** (**overview**).
    - **Elimina stopwords** en **español** e **inglés**.
    - **Prioriza palabras más** frecuentes.
    - Genera hasta top_**n=8 keywords**.

4.4. Reglas de géneros

- ***combinacion_prohibida***(**generos**)
  Revisa si un conjunto de géneros pertenece a una **combinación prohibida** (ej: "comedia + documental").
- ***genero_valido_fuzzy***(generos_item, generos_objetivo)
  Verifica **coincidencia entre géneros de candidato** y **géneros objetivo**.
    - Da prioridad a **géneros fuertes** (animación, ciencia ficción, acción).
    - **Evita falsos positivos con géneros genéricos** (drama, comedia).
    - Aplica **fuzzy matching si no hay coincidencia exacta**.

4.5. Palabras clave especiales

- Se apoyan en ***PALABRAS_CLAVE_ESPECIAL*** (Pixar, Marvel, Anime, Disney, etc.).
  Funciones clave:
- ***Contiene palabra_clave_especial***(texto) → **detecta** si un **overview** o **keywords** contienen palabras especiales con **sinónimos** de inglés y **español** para mejorar la semántica.
- ***grupo_especial_por_query(***query) → **detecta** si la **búsqueda pertenece** a un **grupo/franquicia especial**.
- ***es_palabra_clave_especial***(titulo) → **comprueba** si el título **mismo** es parte de una **palabra clave**.

Esto permite, por ejemplo, que al buscar "**Pixar**", recomiende películas de **Pixar aunque no tengan relación textual directa.**

4.6. Obtención de candidatos

- ***buscar_grupo_en_api(titulos***_grupo, genero_obj_map, tipos)
  Llama a **cache.buscar_en_api_externa** para **traer candidatos** de las **APIs externas** basados en varias **keywords**.
- ***obtener_candidatos_rapidos***(genero, tipo, cantidad)
  Devuelve recomendaciones rápidas predefinidas (ej: siempre devolver "Toy Story" si buscas Pixar). **Esto segura variedad dentro de la misma temática**.

4.7. Enriquecimiento de datos

- ***enriquecer_candidato_rapido***(c)
  Si un candidato **no tiene info completa**, la **busca** en la **caché** o en la **API** externa.
  Completa campos como: **géneros**, **overview**, **keywords**, **año**, **poster**, **director**, **productora**.

4.8. Filtrado y Scoring

- ***enriquecer_y_filtrar***(candidatos, query, genero_obj_map, top_k, ...)
  Es la función más **importante**:
  - **Enriquecer**: completa datos faltantes con cache/API.
  - **Filtrar**:
    - Elimina títulos basura ("Untitled", "Spin-Off").
    - Quita combinaciones prohibidas de géneros.
    - Descartar obras muy antiguas (<1980).
  - **Scoring**: (Explicado en el punto 2.5)

Devuelve una lista **final ordenada por relevancia**.

4.9. Recomendaciones por saga

- ***recomendaciones_sagas***(query, tipo_destino, df, n)
  Busca en el **dataset local** si hay varias **películas/series/libros** de la misma **saga**.
- ***recomendaciones_sagas_api***(query, tipo_destino, n)
  Hace lo mismo, pero con la **API** externa (ej: todas las pelis de ***Harry Potter***).

4.10. Motor principal

- ***buscar_recomendaciones_con_api***(...)
  El **corazón del recomendador**.
  Integra todos los pasos anteriores:
  1. Normaliza y traduce query.
  2. Detecta palabras clave especiales y géneros forzados.
  3. Busca candidatos en APIs externas.
  4. Agrega sagas y recomendaciones rápidas.
  5. Filtra, enriquece y puntúa.
  6. Devuelve los **top_k mejores candidatos**.

- ***buscar_recomendaciones*** (Wrapper público):
  - Llama a **buscar_recomendaciones_con_api.**
  - **Añade enriquecimiento** final con **keywords/overview**.
  - **Devuelve** el **resultado** al **cliente**.

5. Explicación de cache.py

Este archivo gestiona la **caché y comunicación con APIs externas**.
Es clave para reducir la latencia y evitar que tu sistema haga demasiadas llamadas a TMDb, Google Books o RAWG.

Funciones y responsabilidades principales

5.1. Configuración

- Define rutas de caché (CACHE_DIR), tiempo de vida de la caché (CACHE_TTL_SECONDS = 24h) y claves de API.
- Variables de entorno:
    - ***TMDB_API_KEY***
    - ***GOOGLE_API_KEY***
    - ***RAWG_API_KEY***

5.2. Gestión de caché

- ***get_cache_path***(titulo, tipo) → genera la ruta de archivo de caché para un título/tipo.
- ***guardar_cache***(titulo, tipo, data) → guarda resultados en disco.
- ***cargar_cache***(titulo, tipo) → lee resultados guardados, verificando si expiraron.
- ***obtener_info_externa_cache(***titulo, tipo) → usa primero caché, si no existe llama a API y guarda resultado.

Esto significa que si pides *Interstellar* dos veces en un día, la segunda vez ya no consulta la API, sino que lee de disco.

5.3. Llamadas a APIs externas

- ***llamada_api_externa(***titulo, tipo) → decide qué API usar según el tipo:
    - pelicula → TMDb (detalles de película).
    - serie → TMDb (detalles de serie).
    - libro → Google Books.
    - videojuego → RAWG.

Cada llamada devuelve:
título, overview, géneros, autores/directores, año, poster, productora.

5.4. Buscar en APIs (función principal)

- ***buscar_en_api_externa***(query, generos_obj, tipos, limit, df_local, saga)
  Es el **buscador externo universal**.
    - Según el tipo (pelicula, serie, libro, videojuego) llama a la API correcta.
    - Filtra resultados que no coincidan en géneros o keywords.
    - Integra también dataset local si está disponible.
    - Soporta búsqueda de **sagas completas** (ej: *Avengers Collection* en TMDb).

5.5. Funciones extra

- ***generos_expandibles***(generos) → expande un género a otros relacionados (ej: "sci-fi" → también cuenta como "ciencia ficción", "aventura").
- ***to_int_year***(valor) → convierte string/fecha en un año válido.
- ***es_palabra_clave_especial(***titulo) → detecta si un título pertenece a un grupo/franquicia especial (usa fuzzy matching).

Resumen de las dos funciones principales

- **recomendador.py**:
  Es el **motor de inteligencia** → normaliza, busca candidatos, enriquece, aplica reglas, filtra y puntúa.
- **cache.py**:
  Es el **gestor de datos externos** → maneja la caché, llama a APIs externas, normaliza géneros y devuelve datos limpios al motor.

Juntos hacen que el **Sistema de Recomendaciones**:

1. Sea **rápido** (gracias al caché).
2. Sea **inteligente** (gracias a reglas, fuzzy matching y scoring).
3. Devuelva resultados **útiles y filtrados** (no basura).

6. Flujo de una Búsqueda

1. El usuario introduce un título en el frontend.

2. El backend normaliza la consulta y detecta palabras clave especiales.
3. Se consultan primero los géneros forzados o bien la API (cacheada).
4. Se buscan candidatos en:
    - Dataset local (si existe).
    - APIs externas (TMDb, RAWG, Google Books).
5. Se enriquece la información de candidatos con overview, géneros, keywords y poster.
6. Se aplican filtros: duplicados, combinaciones prohibidas, títulos basura.
7. Se priorizan secuelas, recomendaciones rápidas y coincidencia semántica.
8. Se devuelve un JSON con los resultados al frontend.

Idealización