Prácticas de Recuperación de Información

Curso 2024-2025

1 Búsqueda híbrida (y un poco de RAG)

En el *notebook* anterior, utilizamos ChromaDB y *word embedding* para realizar una búsqueda semántica sobre la colección LISA.

Este enfoque abordó parcialmente el problema del desajuste de vocabulario y fue bastante eficaz en la recuperación de información interlingüística. Sin embargo, dado que los materiales de entrenamiento utilizados para crear el modelo de *embedding* preentrenado eran significativamente diferentes del contenido de la colección LISA, tuvo un rendimiento inferior a BM25 con las consultas originales de la colección.

En las diapositivas del curso y los apuntes, analizamos otros escenarios en los que la búsqueda semántica tiene limitaciones que se pueden abordar utilizando modelos léxicos. También tratamos cómo, en algunos casos, la solución es realizar tanto una búsqueda léxica como una búsqueda semántica y luego combinar los resultados.

En este *notebook*, presentaremos esta búsqueda "híbrida" utilizando un enfoque sencillo: promediar las ordenaciones de los documentos de ambas listas. Un enfoque un poco más complejo implicaría utilizar un *reranker* neuronal (aplicarás ese enfoque para desarrollar tu entregable).

Al final, como bonus track, tienes una breve introducción a la Generación Aumentada por Recuperación (RAG).

2 Configuración

Necesitamos instalar los paquetes necesarios para las búsquedas léxica y semántica:

```
# To perform lexical search
pip install bm25s[full]

# To perform semantic search
pip install chromadb
pip install sentence_transformers
```

3 Preparar e indexar el corpus

Necesitamos prepara el corpus para dos enfoques diferentes: bm25s y ChromaDB con *embeddings*. Sin embargo, primero necesitamos descargarlo y descomprimirlo. Posteriormente, parseamos lisa-corpus.json.

Descargamos el fichero LISA-collection.zip de
https://drive.google.com/uc?id=1dIKh0BWK1yyEkGEfUTtx7wPu1LDB8-Sc

```
import json

with open("lisa-corpus.json", "r", encoding="utf-8") as f:
    corpus_content = f.read()
    corpus_content = json.loads(corpus_content)
```

3.1 Preparar el corpus para bm25s

Necesitamos preparar nuestro corpus para BM25S en dos formatos:

1. Una versión literal que mantenga la estructura original de los "documentos".
2. Una versión de texto sencillo para tokenización e indexación.

```
corpus_verbatim = list()
corpus_plaintext = list()
for entry in corpus_content:
    # Nota: nuestros documentos tienen 'id', 'title' y 'content'
    # Vamos a concatenar title y content porque nos interesa para el
    # indexado
    document = {"id": entry["id"], "title": entry["title"].lower(),
        "text": entry["content"].lower()}
    corpus_verbatim.append(document)
    corpus_plaintext.append(f"{entry['title'].lower()}
        {entry['content'].lower()}")
```

Vamos a procesar el texto. Para ello, aplicamos un *stemmer* para el idioma inglés y eliminamos palabras vacías al realizar la tokenización:

```
import bm25s # Para crear índices y buscar documentos con BM25
import Stemmer  # Para estematizar términos

stemmer = Stemmer.Stemmer("english")
corpus_tokenized = bm25s.tokenize(corpus_plaintext, stopwords="en",
    stemmer=stemmer, show_progress=True)
```

3.2 Indexar el corpus con bm25s

Ahora crearemos nuestro recuperador basado en BM25 e indexaremos el corpus tokenizado.

```
# Escogemos lucene, pero puedes cambiarla y probar otras opciones
bm25_flavor = "lucene"
idf_flavor = "lucene"

retriever = bm25s.BM25(corpus=corpus_verbatim, method=bm25_flavor,
    idf_method=idf_flavor)
retriever.index(corpus_tokenized, show_progress=True)

retriever.save("LISA", corpus=corpus_verbatim)
```

3.3 Preparar el corpus para ChromaDB

Necesitamos el texto de los documentos, sus identificadores y sus *embeddings*.

```
# Prepare documents for ChromaDB
chromadb_documents = corpus_plaintext
chromadb_doc_ids = []

for document in corpus_content:
  doc_id = str(document["id"])
  chromadb_doc_ids.append(doc_id)
```

En vez de crear los *embeddings*, vamos a cargar los precalculados que descargamos de https://drive.google.com/uc?id=11Z4fzljPrY8w7tDysNjR3zM4sme9pMDD.

```
# Debes ejecutar esta sentencia para crear los embeddings…
# chromadb_embeddings = model.encode(chromadb_documents, batch_size=100,
#     show_progress_bar=True) #, device='cuda')

import pickle

with open("LISA-embeddings.pickle", "rb") as f:
  chromadb_embeddings = pickle.load(f)
```

3.4 Almacenar la colección en ChromaDB

Ya podemos incorporar el texto de los documentos, los identificadores y los embeddings a ChromaDB; pero, primero, necesitamos crear la colección.

```
import chromadb
from chromadb.utils import embedding_functions
from sentence_transformers import SentenceTransformer

# Inicializar el modelo del transformer de sentencias
model = SentenceTransformer(
    'sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2')

# Crear un cliente de ChromaDB persistente
client = chromadb.PersistentClient(path="./chromadb-storage/")

# Creamos la colección. Ten en cuenta cómo proporcionamos el modelo
# preentrenado de embedding (este es el modelo multilingüe) y cómo
# especificamos la métrica de distancia para encontrar los vecinos más
# cercanos
collection = client.create_collection(
  name="LISA_collection",
  embedding_function=embedding_functions.SentenceTransformerEmbeddingFunc
tion(model_name="sentence-transformers/paraphrase-multilingual-MiniLM-
L12-v2"),
  metadata={"hnsw:space": "cosine"}
     # https://docs.trychroma.com/guides#changing-the-distance-function
)
```

Definimos una función para generar lotes (recuerda el *notebook* anterior):

```
# Función para dividir una lista (de documentos,ids, embeddings) en lotes
# porque ChromaDB puede dar problemas al intentar añadir un dataset grande
# a la colección de una sola vez
def get_batches(lista, chunk_size=100):
    return [lista[i:i + chunk_size] for i in range(0, len(lista),
        chunk_size)]
```

Ahora, utilizamos la función anterior para ir creando la colección por lotes:

```
document_batches = get_batches(chromadb_documents)
ids_batches = get_batches(chromadb_doc_ids)
embedding_batches = get_batches(chromadb_embeddings)

for i in range(len(document_batches)):
  documents = document_batches[i]
  doc_ids = ids_batches[i]
  embeddings = embedding_batches[i]

  # Add the documents, ids and embeddings to the collection
  collection.add(
    documents=documents,
    ids=doc_ids,
    embeddings=embeddings
```

4 Buscar con bm252 y ChromaDB

4.1 Cargar las consultas a enviar

Vamos a cargar todas las consultas: lisa-queries.json, lisa-queries-paraphrased.json y lisa-queries-spanish.json.

```
import json

all_queries = dict()

for filename in ["lisa-queries.json", "lisa-queries-paraphrased.json",
    "lisa-queries-spanish.json"]:
    with open(filename, "r", encoding="utf-8") as f:
        content = f.read()
        data = json.loads(content)
        all_queries[filename] = data
print(all_queries)
```

Vamos a crear una función para enviar las consultas al motor léxico:

```
# Función para cargar consultas desde un fichero json para enviar a una
# colección dada y obtener listas de resultados.
def submit_queries_to_lexical_and_get_run(queries, stemmer, retriever,
    max_results=10):

    run = {}

    # Iteramos para enviar cada consulta al recuperador para obtener
    # resultados
    for query in queries:
        # Para evaluar después el rendimiento, es útil conservar el id de
        # la consulta
        query_id = query["id"]

        query_string = query["query"].lower()

        # Tokenizamos la consulta. ¡Atención! Debemos tokenizar la consulta
        # con la misma configuración que usamos al indexar el corpus. En
        # este caso, eliminando las palabras vacías del inglés y aplicando
        # el stemmer de inglés
        #
        query_tokenized = bm25s.tokenize(query_string, stopwords="en",
            stemmer=stemmer, show_progress=False)

        # Retornamos los k resultados más "top" como una tupla con nombre.
        # Por favor, lee la documentación de BM25S para otras alternativas.
        #
        results = retriever.retrieve(query_tokenized,
            corpus=retriever.corpus, k=max_results, return_as="tuple",
            show_progress=False)

        # Los documentos y sus puntuaciones se guardan en dos campos
        # diferentes de la tupla.
        #
        returned_documents = results.documents[0]
        relevance_scores = results.scores[0]
```

```
        returned_ids = []
        for i in range(len(returned_documents)):
##          print(f"\t{i}\t{relevance_scores[i]}\t{
##              returned_documents[i]['id']}\t
##              {returned_documents[i]['title'][0:80]}...")
            returned_ids.append(str(returned_documents[i]["id"]))
        run[query_id] = returned_ids

    return run
```

4.2 Conectar los motores de búsqueda léxico y semántico

Recuerda que indexar (o almacenar *embeddings* en una base de datos vectorial) es un paso separado y previo a la recuperación, aunque estemos haciendo ambas en el mismo *notebook*.

4.2.1 Conectar con el motor de búsqueda léxico

```
import bm25s # Para crear índices y buscar documentos con BM25
import Stemmer # Para estematizar términos

retriever = bm25s.BM25.load("LISA", load_corpus=True)
stemmer = Stemmer.Stemmer("english")
```

4.2.2 Enviar las consultas al motor de búsqueda léxico

Vamos a ejecutar tres versiones de las consultas: originales, parafraseadas y en español

Consultas originales enviadas al motor de búsqueda léxico

```
original_queries = all_queries["lisa-queries.json"]
original_lexical_run = submit_queries_to_lexical_and_get_run(
    original_queries, stemmer, retriever)
```

Consultas parafraseadas enviadas al motor de búsqueda léxico

```
paraphrased_queries = all_queries["lisa-queries-paraphrased.json"]
paraphrased_lexical_run = submit_queries_to_lexical_and_get_run(
    paraphrased_queries, stemmer, retriever)
```

Consultas en español enviadas al motor de búsqueda léxico

```
spanish_queries = all_queries['lisa-queries-spanish.json']
spanish_run = submit_queries_to_lexical_and_get_run(spanish_queries,
    stemmer, retriever)
```

4.2.3 Conectar con el motor de búsqueda semántico

Vamos a conectarnos con la colección que queremos, LISA_collection, en ChromaDB:

```
import chromadb
from chromadb.utils import embedding_functions
from sentence_transformers import SentenceTransformer

# Crear un cliente ChromaDB persistente
client = chromadb.PersistentClient(path="./chromadb-storage/")

# Obtener la colección disponible en ChromaDB
existing_collections = client.list_collections()

collection_name = "LISA_collection"

# Inicializar el modelo del transformer de sentencias
```

```
model = SentenceTransformer(
    'sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2')

# Comprobar si la colección existe
if collection_name in [col.name for col in existing_collections]:
    # No tiene mucho sentido que tengamos que indicar qué función de
    # embedding usa la colección, *pero* si no se indica explícitamente,
    # ChromaDB utilizará la función de embedding por defecto y será como
    # comparar manzanas con naranjas…
  collection = client.get_collection(
        collection_name,
        embedding_function=embedding_functions.SentenceTransformerEmbeddi
ngFunction(model_name="sentence-transformers/paraphrase-multilingual-
MiniLM-L12-v2")
    )

    existing_ids = collection.get()["ids"]
    print(f"The collection {collection_name} contains {len(existing_ids)}
        documents")
else:
  print(f"{collection_name} doesn't exist! You need to create it.")
```

4.2.4 Enviar consultas al motor de búsqueda semántico

Vamos a enviar los tres tipos de consultas también a la colección que está en ChromaDB.

Vamos a definir primero la función para enviar las consultas:

```
# Función para cargar consultas desde un fichero json para enviar a una
# colección dada y obtener listas de resultados.
def submit_queries_to_semantic_and_get_run(queries, collection,
    max_results=10):
    # Inicializar el diccionario run
    run = {}

    # Procesar cada consulta
    for query in queries:
        query_id = query["id"]
        query_text = query["query"].lower()

        # Enviar la consulta de la colección y obtener los resultados
        results = collection.query(
            query_texts=[query_text],
            n_results=max_results
        )

        # Guardar los resultados en el diccionario run
        run[query_id] = results['ids'][0]

    return run
```

Consultas originales enviadas al motor de búsqueda semántico

```
original_queries = all_queries["lisa-queries.json"]
original_semantic_run = submit_queries_to_semantic_and_get_run(
    original_queries, collection)
```

Consultas parafraseadas enviadas al motor de búsqueda semántico

```
paraphrased_queries = all_queries["lisa-queries-paraphrased.json"]
```

```
paraphrased_semantic_run = submit_queries_to_semantic_and_get_run(
    paraphrased_queries, collection)
```

Consultas en español enviadas al motor de búsqueda semántico

```
spanish_queries = all_queries['lisa-queries-spanish.json']
spanish_semantic_run = submit_queries_to_semantic_and_get_run(
    spanish_queries, collection)
```

5 Combinación de resultados léxicos y semánticos

Aunque existen muchas formas de combinar múltiples listas de resultados de diferentes sistemas, existe un método simple, robusto y ampliamente utilizado: RRF o Reciprocal Rank Fusion (Cormack *et al.* 2009). Otra opción, que explorará como parte del producto final, es la reclasificación.

$$RRFscore(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

Veamos cómo funciona con un ejemplo…

Primero tenemos las 10 mejores películas según valoraciones en Rotten Tomatoes:

1. L.A. Confidential (1997)
2. The Godfather (1972) - El Padrino
3. Casablanca (1942)
4. Seven Samurai (1954) - Los siete samuráis
5. Parasite (2019) - Parásitos
6. Schindler's List (1993) - La lista de Schindler
7. Top Gun: Maverick (2022)
8. Toy Story 2 (1999)
9. Chinatown (1974)
10. On the Waterfront (1954) - Nido de ratas

Después tenemos las 10 mejores según valoraciones en IMDb:

1. The Shawshank Redemption (1994) - Cadena perpetua
2. The Godfather (1972) - El Padrino
3. The Dark Knight (2008) - El caballero oscuro
4. The Godfather Part II (1974) - El Padrino: Parte II
5. 12 Angry Men (1957) - 12 hombres sin piedad
6. LotR: The Return of the King (2003) - ESDLA: El retorno del Rey
7. Schindler's List (1993) - La lista de Schindler
8. Pulp Fiction (1994)
9. LotR: The Fellowship of the Ring (2001) - ESDLA: La Comunidad del Anillo
10. The Good, the Bad and the Ugly (1966) - El bueno, el feo y el malo

“L.A. Confidential” es la primera película en la lista de Rotten Tomatoes, pero no aparece entre las 10 de IMDb así que se asume un valor nulo.

En consecuencia, su RRF sería 1/(60+1) + 0, es decir, 0,0164.

“El Padrino” aparece en ambas listas en segundo lugar, así que tendría un valor RRF de 1/(60+2) + 1/(60+2) = 0,0323.

Los valores de RRF del resto de películas se calcularían de manera acorde y se obtendría la siguiente lista común…

1. **The Godfather (1972) - El Padrino … 0.0323**
2. **Schindler's List (1993) - La lista de Schindler … 0.0301**
3. **L.A. Confidential (1997) … 0.0164**
4. **The Shawshank Redemption (1994) - Cadena perpetua … 0.0164**
5. **Casablanca (1942) … 0.0159**
6. **The Dark Knight (2008) - El caballero oscuro … 0.0159**
7. **Seven Samurai (1954) - Los siete samuráis … 0.0156**
8. **The Godfather Part II (1974) - El Padrino: Parte II … 0.0156**
9. **12 Angry Men (1957) - 12 hombres sin piedad … 0.0154**
10. **Parasite (2019) - Parásitos … 0.0154**
11. LotR: The Return of the King (2003) - ESDA: El retorno del Rey … 0.0152
12. Top Gun: Maverick (2022) … 0.0149
13. Pulp Fiction (1994) … 0.0147
14. Toy Story 2 (1999) … 0.0147
15. Chinatown (1974) … 0.0145
16. LotR: The Fellowship of the Ring (2001) - ESDA: La Comunidad del Anillo … 0.0145
17. On the Waterfront (1954) - Nido de ratas … 0.0143
18. The Good, the Bad and the Ugly (1966) - El bueno, el feo y el malo … 0.0143

El código Python para calcular RRF es el siguiente:

```
def reciprocal_rank_fusion(ranking1, ranking2, k=60):
    fused_rank = dict()

    ranking1 = {string: index for index, string in enumerate(ranking1)}
    ranking2 = {string: index for index, string in enumerate(ranking2)}

    all_docs_ids = set(ranking1.keys()).union(set(ranking2.keys()))

    rrf_values = dict()

    for doc_id in all_docs_ids:
        try:
            rank_1 = 1/(k + ranking1[doc_id])
        except:
            rank_1 = 0
        try:
            rank_2 = 1/(k + ranking2[doc_id])
        except:
            rank_2 = 0

        rrf_values[doc_id] = rank_1 + rank_2

    sorted_rrf = sorted(rrf_values.items(), key=lambda x: x[1],
reverse=True)

    return sorted_rrf

top_10_rotten_tomatoes = ["L.A. Confidential (1997)", "The Godfather
(1972) - El Padrino", "Casablanca (1942)", "Seven Samurai (1954) - Los
siete samuráis", "Parasite (2019) - Parásitos", "Schindler's List (1993)
```

```
- La lista de Schindler", "Top Gun: Maverick (2022)", "Toy Story 2 (1999)",
"Chinatown (1974)", "On the Waterfront (1954) - Nido de ratas"]

top_10_imdb = ["The Shawshank Redemption (1994) - Cadena perpetua", "The
Godfather (1972) - El Padrino", "The Dark Knight (2008) - El caballero
oscuro", "The Godfather Part II (1974) - El Padrino: Parte II", "12 Angry
Men (1957) - 12 hombres sin piedad", "LotR: The Return of the King (2003)
- ESDLA: El retorno del Rey", "Schindler's List (1993) - La lista de
Schindler", "Pulp Fiction (1994)", "LotR: The Fellowship of the Ring (2001)
- ESDLA: La Comunidad del Anillo", "The Good, the Bad and the Ugly (1966)
- El bueno, el feo y el malo"]

top_movies = reciprocal_rank_fusion(top_10_rotten_tomatoes, top_10_imdb)

print(top_movies[:10])

[('The Godfather (1972) - El Padrino', 0.03278688524590164), ("Schindler's
List (1993) - La lista de Schindler", 0.030536130536130537), ('L.A.
Confidential (1997)', 0.016666666666666666), ('The Shawshank Redemption
(1994) - Cadena perpetua', 0.016666666666666666), ('The Dark Knight (2008)
- El caballero oscuro', 0.016129032258064516), ('Casablanca (1942)',
0.016129032258064516), ('Seven Samurai (1954) - Los siete samuráis',
0.015873015873015872), ('The Godfather Part II (1974) - El Padrino: Parte
II', 0.015873015873015872), ('Parasite (2019) - Parásitos', 0.015625),
('12 Angry Men (1957) - 12 hombres sin piedad', 0.015625)]
```

¡Atención! Las diferencias en las puntuaciones RRF surgen porque el ejemplo comienza los *rankings* en la posición 1, mientras que la función `enumerate` de Python comienza en 0. Sin embargo, la lista fusionada mantiene el mismo orden general. (Nota: En caso de empates, el orden puede variar).

```
def mix_lexical_semantic_runs(lexical_run, semantic_run):
    mixed_run = {}
    for query_id in lexical_run.keys():
      max_results = len(lexical_run[query_id])
      lexical_results = lexical_run[query_id]
      semantic_results = semantic_run[query_id]

      lexical_results_dict = {string: index for index, string in
enumerate(lexical_results)}
      semantic_results_dict = {string: index for index, string in
enumerate(semantic_results)}

      lexical_semantic_results =
reciprocal_rank_fusion(lexical_results_dict, semantic_results_dict)

      mixed_run[query_id] = list(lexical_semantic_results)[:max_results]
      mixed_run[query_id] = [item[0] for item in mixed_run[query_id]]

    return mixed_run
```

Ahora ejecutamos las funciones anteriores para combinar los resultados de los tres tipos de consultas:

```
original_mixed_run = mix_lexical_semantic_runs(original_lexical_run,
    original_semantic_run)
paraphrased_mixed_run =
    mix_lexical_semantic_runs(paraphrased_lexical_run,
```

```
    paraphrased_semantic_run)
spanish_mixed_run =
    mix_lexical_semantic_runs(spanish_run, spanish_semantic_run)
```

6 Evaluar el rendimiento

Como hicimos en los *notebooks* anteriores, vamos a calcular el rendimiento del enfoque híbrido.

```
# Función para calcular micro y macropromedios de la precisión,
# exhaustividad y F1-score
def compute_precision_recall_f1(run, relevance_judgements):
    # Inicializar listas para guardar la precisión, exhaustividad y
    # F1-score de cada consulta
    precision_values = []
    recall_values = []
    f1_values = []

    # Initializar contadores globales para micropromedios
    global_retrieved = 0
    global_relevant = 0
    global_retrieved_and_relevant = 0

    # Calcular precisión, exhaustividad y F1-score para cada consulta
    for query_id in run.keys():
        retrieved_results = run[query_id]
        relevant_results = relevance_judgements[query_id]
        relevant_and_retrieved = set(retrieved_results) &
            set(relevant_results)

        # Actualizar contadores globales
        global_retrieved += len(retrieved_results)
        global_relevant += len(relevant_results)
        global_retrieved_and_relevant += len(relevant_and_retrieved)

        # Calcular precisión y exhaustividad
        precision = len(relevant_and_retrieved) / len(retrieved_results)
            if len(retrieved_results) > 0 else 0
        recall = len(relevant_and_retrieved) / len(relevant_results)
            if len(relevant_results) > 0 else 0

       # Calcular F1-score si la precisión y la exhaustividad no son cero
        if (precision + recall) > 0:
            f1 = 2 * (precision * recall) / (precision + recall)
            f1_values.append(f1)

        # Añadir precisión y exhaustividad para la consulta actual
        precision_values.append(precision)
        recall_values.append(recall)

    # Calcular macropromedios
    macro_average_precision = sum(precision_values) /
        len(precision_values) if precision_values else 0
    macro_average_recall = sum(recall_values) / len(recall_values)
       if recall_values else 0
    macro_average_f1 = sum(f1_values) / len(f1_values)
       if f1_values else 0

    # Mostrar los macropromedios
```

```
    print(f"Macro-averaged Precision:{round(macro_average_precision,3)}")
    print(f"Macro-averaged Recall: {round(macro_average_recall,3)}")
    print(f"Macro-averaged F1: {round(macro_average_f1,3)}")
    print("")

    # Calcular micropromedios
    micro_average_precision = global_retrieved_and_relevant /
        global_retrieved if global_retrieved > 0 else 0
    micro_average_recall = global_retrieved_and_relevant / global_relevant
        if global_relevant > 0 else 0
    micro_average_f1 = (2 * (micro_average_precision *
        micro_average_recall) / (micro_average_precision +
        micro_average_recall))
        if (micro_average_precision + micro_average_recall) > 0 else 0

    # Mostrar micropromedios
    print(f"Micro-averaged Precision:{round(micro_average_precision,3)}")
    print(f"Micro-averaged Recall: {round(micro_average_recall,3)}")
    print(f"Micro-averaged F1: {round(micro_average_f1,3)}")
```

6.1 Cargar los juicios de relevancia

Solo podemos comprobar el rendimiento si tenemos juicios de relevancia.

```
# Cargamos los juicios de relevancia
with open("lisa-relevance-judgements.json", "r", encoding="utf-8") as f:
    relevance_judgements = json.load(f)

# El formato de los juicios de relevancia necesita ser ligeramente
# transformado
relevance_judgements_reformat = dict()
for entry in relevance_judgements:
  query_id = entry["query_id"]
  rel_docs = entry["rel_docs"].split(", ")
  relevance_judgements_reformat[query_id] = rel_docs
```

6.2 Rendimiento cuando se utilizan las consultas originales

```
compute_precision_recall_f1(original_mixed_run,
    relevance_judgements_reformat)
```

Recuerda que el rendimiento que obtuvimos cuando utilizamos BM25S con la implementación lucene para la función de ranking y el cálculo del IDF fue:

Macro-averaged Precision: 0.303
Macro-averaged Recall: 0.394
Macro-averaged F1: 0.332
Micro-averaged Precision: 0.303
Micro-averaged Recall: 0.28
Micro-averaged F1: 0.291

Y el rendimiento utilizando *word embedding* y el coseno como distancia en ChromaDB fue:

Macro-averaged Precision: 0.223
Macro-averaged Recall: 0.264
Macro-averaged F1: 0.256
Micro-averaged Precision: 0.223

Micro-averaged Recall: 0.206
Micro-averaged F1: 0.214

¿Cuál será el rendimiento del enfoque híbrido?

Desafortunadamente, el enfoque híbrido no fue capaz de mejorar los resultados de BM25:

Macro-averaged Precision: 0.28
Macro-averaged Recall: 0.371
Macro-averaged F1: 0.309

Micro-averaged Precision: 0.28
Micro-averaged Recall: 0.259
Micro-averaged F1: 0.269

6.3 Rendimiento cuando se utilizan las consultas parafraseadas

```
compute_precision_recall_f1(paraphrased_mixed_run,
    relevance_judgements_reformat)
```

Recuerda que el rendimiento que obtuvimos cuando utilizamos BM25S con la implementación lucene para la función de ranking y el cálculo del IDF fue:

Macro-averaged Precision: 0.143
Macro-averaged Recall: 0.208
Macro-averaged F1: 0.213
Micro-averaged Precision: 0.143
Micro-averaged Recall: 0.132
Micro-averaged F1: 0.137

Y el rendimiento utilizando búsqueda semántica fue:

Macro-averaged Precision: 0.166
Macro-averaged Recall: 0.179
Macro-averaged F1: 0.236
Micro-averaged Precision: 0.166
Micro-averaged Recall: 0.153
Micro-averaged F1: 0.159

¡Veamos el rendimiento del enfoque híbrido!

En este caso, las consultas parafraseadas y la búsqueda híbrida han mejorado ligeramente los resultados de la búsqueda léxica, pero no de la búsqueda semántica...

Macro-averaged Precision: 0.166
Macro-averaged Recall: 0.201
Macro-averaged F1: 0.213

Micro-averaged Precision: 0.166
Micro-averaged Recall: 0.153
Micro-averaged F1: 0.159

6.4 Rendimiento cuando se utilizan las consultas en español

```
compute_precision_recall_f1(spanish_mixed_run,
    relevance_judgements_reformat)
```

Como recordarás, la búsqueda léxica simplemente no puede manejar la recuperación de información interlingüística, y el rendimiento fue prácticamente cer0.

Sin embargo, el rendimiento de la búsqueda semántica fue bastante bueno:

Macro-averaged Precision: 0.169
Macro-averaged Recall: 0.21
Macro-averaged F1: 0.221
Micro-averaged Precision: 0.169
Micro-averaged Recall: 0.156
Micro-averaged F1: 0.162

En este caso, solo podemos asumir que el rendimiento de la búsqueda híbrida empeorará respecto a la búsqueda semántica.

Como esperábamos, el rendimiento de la búsqueda híbrida es peor, debido a la extremadamente mala precisión y exhaustividad de la búsqueda léxica en el marco interlingüístico:

Macro-averaged Precision: 0.097
Macro-averaged Recall: 0.149
Macro-averaged F1: 0.172

Micro-averaged Precision: 0.097
Micro-averaged Recall: 0.09
Micro-averaged F1: 0.093

7 Conclusión

Este *notebook* ilustra cómo la búsqueda híbrida puede combinar resultados de modelos de recuperación de información léxicos y semánticos. Este pequeño experimento ha demostrado el potencial y las limitaciones de la búsqueda híbrida.

Cuando los modelos léxicos y semánticos aportan sus puntos fuertes, su combinación puede superarlos a ambos. Sin embargo, la búsqueda híbrida no siempre es la solución óptima. En los casos en los que un método supera significativamente al otro (como vimos con BM25 superando a la búsqueda semántica con las consultas originales, o la búsqueda semántica manejando consultas interlingüísticas), el enfoque híbrido puede no mejorar los resultados.

En resumen, la búsqueda híbrida es más eficaz cuando ambos componentes proporcionan información complementaria y valiosa. Antes de implementar un sistema híbrido, es crucial evaluar las características de los datos, los tipos de consultas típicos y el rendimiento individual de cada modelo. Si bien la búsqueda híbrida puede ser una herramienta poderosa en la recuperación de información, debe verse como uno de los muchos enfoques, que se deben utilizar según las necesidades específicas de cada escenario.

8 Bonus track: Generación Automática por Recuperación

La Generación Aumentada por Recuperación (RAG) consiste en utilizar un *first-stage retriever* para obtener una lista de resultados relevantes (documentos o pasajes) y luego utilizar esos textos y la consulta original para crear un *prompt* para un LLM.

Para implementar un RAG se necesitará acceder a un API del LLM o desplegar un localmente (por ejemplo, utilizando OpenLlama). Dado que ambos escenarios están fuera del alcance de esa asignatura, realizaremos una demo muy simple:

1. Obtendremos los resultados mixtos para una sola consulta, es decir, la combinación de búsqueda léxica y semántica.
2. Obtendremos los contenidos textuales de esos resultados.
3. Combinaremos los contenidos textuales y la consulta en un *prompt* significativo.
4. Puedes copiar y pegar ese *prompt* en cualquier chatbot como ChatGPT o Claude.

Puedes consultar los resultados obtenidos con algunos chatbots en https://docs.google.com/document/d/1skqSnu8bz1p8IBaMUjAUc1dcquZzYY4BoDqcsjifToA/edit?tab=t.0#heading=h.szlf5wio0fc4

```
with open("lisa-corpus.json", "r", encoding="utf-8") as f:
    corpus_content = f.read()
    corpus_content = json.loads(corpus_content)

query_id = 18

query = original_queries[query_id-1]["query"].lower()

query_mixed_results = original_mixed_run[query_id]

doc_texts = []
for doc_id in query_mixed_results:
  document = corpus_content[int(doc_id)]
  doc_text = f"- Document Identifider: {doc_id}. Title: {document['title']}
      Content: {document['content']}"
  doc_texts.append(doc_text)

doc_texts = "\n".join(doc_texts)

prompt_template = "You are a highly knowledgeable assistant capable of
    providing accurate and relevant information. I have a query and a set
    of documents that may contain important information related to it.
    \n\n**Query:** <<QUERY>>\n\n**Documents:**\n<<DOC_TEXTS>>\n\nBased on
    the query and the information provided in the documents, please select
    *the most relevant documents* to answer the query, and generate a
    comprehensive two or three paragraph answer that addresses the query,
    incorporating insights from *all the relevant documents* as needed and
    referencing the document identifiers between square brackets.\n"

with open(f"RAG-prompt-{query_id}.txt", "w") as f:
  f.write(prompt_template.replace("<<QUERY>>",
      query).replace("<<DOC_TEXTS>>", doc_texts))

from google.colab import files
files.download(f"RAG-prompt-{query_id}.txt")
```