Prácticas de Recuperación de Información

Curso 2024-2025

1 Búsqueda semántica utilizando bases de datos vectoriales y *word embeddings*

En el *notebook* anterior, utilizamos bm25s para implementar un motor de búsqueda léxica para la colección LISA.

Las limitaciones de los modelos de recuperación de información léxicos eran evidentes: en primer lugar, el rendimiento disminuye cuando hay un desajuste de vocabulario entre las consultas y los documentos; en segundo lugar, la recuperación de información interlingüística no es factible.

Para abordar estos problemas, podemos utilizar modelos que capturan la semántica subyacente de los textos. Estos modelos pueden recuperar documentos relevantes incluso si no contienen ninguno de los términos de la consulta.

Tanto las diapositivas como los apuntes proporcionan una explicación más profunda de dichos modelos. La idea básica subyacente es simple: transformar vectores léxicos dispersos en vectores más densos dentro de un espacio con muchas menos dimensiones que el tamaño del vocabulario.

Uno de los primeros enfoques para lograr esto fue Latent Semantic Indexing, que puedes explorar en el *notebook* "Demostración sencilla de LSI usando SVD". Los enfoques más modernos se basan en el uso de *word embeddings*.

En este *notebook*, utilizaremos una base de datos vectorial, ChromaDB (https://www.trychroma.com/), que puede almacenar las representaciones de *embedding* de los documentos y devolver aquellas que sean más cercanas (y, por lo tanto, más similares o relevantes) a los *embeddings* de una consulta determinada.

Ten en cuenta lo siguiente:

- Estamos utilizando sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 (https://huggingface.co/sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2) como modelo de *embedding* preentrenado. Es bastante bueno teniendo en cuenta su tamaño, los requisitos de memoria y su multilingüismo.
- Otro modelo preentrenado multilingüe que funciona mejor que este, pero es un poco más grande y más costoso computacionalmente, es intfloat/multilingual-e5-large-instruct (https://huggingface.co/intfloat/multilingual-e5-large-instruct).
- Para obtener más detalles sobre la selección de un modelo de *embedding*, consulta "Understanding embedding models: make an informed choice for your RAG" (https://unstructured.io/blog/understanding-embedding-models-make-an-informed-choice-for-your-rag) .

2 Indexar con ChromaDB

En esta parte del *notebook*, crearemos una "colección" en ChromaDB para almacenar las representaciones *embedding* de los documentos de la colección LISA. En el lenguaje de ChromaDB, una colección se puede considerar como algo similar a un "índice" o una "base de datos".

2.1 Configuración

En primer lugar, vamos a instalar los paquetes necesarios de Python:

```
pip install chromadb
```

```
pip install sentence_transformers
```

Ahora, vamos a importar las librerías que necesitamos:

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

2.2 Cargar el corpus

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

Posteriormente, lo descomprimimos y parseamos lisa-corpus.json:

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

2.3 Crear la colección con ChromaDB

Para crear la colección, necesitamos instanciar un cliente. También sería conveniente especificar la función de *embedding* y la métrica de distancia.

En este caso, también estamos guardando la base de datos en disco (se creará la carpeta chromadb-storage).

```
# 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
)
```

Para agregar documentos a la colección, podemos simplemente pasar los textos de los documentos y ChromaDB los transformará en *embeddings*. Sin embargo, dado que ChromaDB no utiliza una GPU, el proceso puede ser lento.

Si tienes acceso a una GPU, puedes crear los *embeddings* utilizando un paquete que aproveche sus beneficios. En este *notebook*, puede solicitar un entorno T4 (siempre que haya recursos disponibles en Google Colab) y modificar el código según sea necesario (asegúrate de prestar atención a las secciones comentadas).

Para esta pequeña colección, la transformación de *embedding* debería llevar menos de 15 minutos.

Además, es posible que ChromaDB no pueda manejar toda la colección a la vez. Por lo tanto, puede ser útil "dividir" los documentos, los identificadores de documentos y los *embeddings* en "lotes".

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

Para ahorrar tiempo, vamos a descargar los *embeddings* precalculados de https://drive.google.com/uc?id=11Z4fzljPrY8w7tDysNjR3zM4sme9pMDD y descomprimimos el fichero.

Ahora vamos a cargar el documento con los *embeddings*:

```
import pickle

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

Ahora que tenemos los *embeddings* de los documentos, podemos comenzar a añadir documentos, identificadores y *embeddings* a la colección en ChromaDB.

En primer lugar, definimos una función para generar lotes:

```
# 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)]
```

A continuación, 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
```

Ten en cuenta que estamos asumiendo que la colección no existe y la estamos creando. Si la colección ya existe y quieres actualizar o modificar documentos existentes (según su identificador), consulta la documentación de ChromaDB (https://docs.trychroma.com/).

3 Buscar con ChromaDB

En esta parte del *notebook*, vamos a buscar una determinada colección en ChromaDB, lanzar consultas y evaluar su rendimiento.

3.1 Configuración

En primer lugar, instalamos los paquetes necesarios:

```
pip install chromadb
```

```
pip install sentence_transformers
```

Ahora, importamos las librerías necesarias:

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

import json
```

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

```
# 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.")
```

La colección LISA en realidad contiene 6000 documentos; sin embargo, uno de ellos está repetido (incluido un identificador duplicado) y esto causa problemas en ChromaDB; por lo tanto, se eliminó del archivo JSON.

3.2 Enviar consultas

Vamos a cargar todas las consultas: lisa-queries.json, lisa-queries-paraphrased.json y lisa-queries-spanish.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 y obtener los resultados asociados a ellas, para posteriormente calcular el rendimiento (precisión, exhaustividad y F1-score).

Ten en cuenta que, aunque enviemos consultas textuales, ChromaDB las transformará en *embeddings* (utilizando el modelo especificado al crear la conexión) y luego retornará los *embeddings* de la colección más cercanos al *embedding* de la consulta.

```
# Función para cargar consultas desde un fichero json para enviar a una
# colección dada y obtener listas de resultados.
def submit_queries_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
```

3.2.1 Consultas originales de la colección LISA

Estas son las consultas originales en la colección de evaluación; fueron recopiladas por los autores de la colección, por lo que el desajuste de vocabulario entre las consultas y los documentos debería ser mínimo.

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

3.2.2 Consultas parafraseadas

Estas consultas se han parafraseado utilizando un LLM, por lo que deberían transmitir un significado similar a las consultas originales pero el desajuste de vocabulario va a ser mayor, en comparación con dichas consultas originales.

Por ejemplo, la consulta 18 originalmente era:

I WOULD BE PLEASED TO RECEIVE ANY INFORMATION ON TELECOMMUNICATIONS NETWORKS. THIS COULD INCLUDE BOTH WIDE AREA NETWORKS (MESH) AND LOCAL AREA NETWORKS (BUS, RING). TELECOMMUNICATIONS, LOCAL AREA NETWORKS, WIDE AREA NETWORKS, DATA TRANSMISSION, DISTRIBUTED PROCESSING, DISTRIBUTED ROUTING, PACKET-SWITCHED NETWORKS, MESH, BUS, RING.

Y se parafraseó como:

I'd like information on how computers talk to each other over long and short distances. This includes big networks that cover wide areas and smaller networks within buildings or campuses.

Los humanos podríamos parafrasear la misma consulta de formas muy diferentes, por lo que te animamos a que pruebes tus propias versiones. Sin embargo, el único propósito de esto es ilustrar las enormes diferencias entre el léxico de las consultas comunes de los usuarios y el de los documentos de la colección.

```
paraphrased_queries = all_queries["lisa-queries-paraphrased.json"]
paraphrased_run = submit_queries_and_get_run(paraphrased_queries,
    collection)
```

3.2.3 Consultas en español

Estas consultas han sido traducidas automáticamente al español, por lo que es muy poco probable que un motor de búsqueda léxica produzca resultados significativos. Por lo tanto, el propósito de este experimento es demostrar que la recuperación de información interlingüística es prácticamente imposible si solo se depende de la búsqueda léxica.

Por ejemplo, la consulta 18 en español es:

ME ENCANTARÍA RECIBIR CUALQUIER INFORMACIÓN SOBRE REDES DE TELECOMUNICACIONES. ESTO PODRÍA INCLUIR TANTO REDES DE ÁREA AMPLIA (MESH) COMO REDES DE ÁREA LOCAL (BUS, RING). TELECOMUNICACIONES, REDES DE ÁREA LOCAL, REDES DE ÁREA AMPLIA, TRANSMISIÓN DE DATOS, PROCESAMIENTO DISTRIBUIDO, ENRUTAMIENTO DISTRIBUIDO, REDES DE CONMUTACIÓN DE PAQUETES, MALLA, BUS, ANILLO.

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

3.3 Evaluar el rendimiento

Vamos a calcular la precisión y la exhaustividad (https://en.wikipedia.org/wiki/Precision_and_recall) y F1-score (https://en.wikipedia.org/wiki/F-score); posteriormente, obtendremos micropromedios y macropromedios (https://danigayo.prof/downloads/blindlight-dgayo-disertacion-glosario.pdf).

```
# 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)}")
```

3.3.1 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
```

3.3.2 Rendimiento cuando se utilizan las consultas originales

```
compute_precision_recall_f1(original_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

El rendimiento utilizando *word embedding* y el coseno como distancia en ChromaDB es:

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

¡Es mucho peor! ¿Cómo puede ser eso?

La razón es simple: la búsqueda semántica basada en *word embedding* tiene un rendimiento inferior cuando se aplica a colecciones de texto que difieren significativamente de los materiales utilizamos para el entrenamiento (el *out-of-domain problem*).

La solución es ajustar los *embeddings* para nuestras colecciones específicas.

Por otro lado, esto demuestra claramente por qué BM25 todavía se considera un competidor robusto, especialmente considerando que es computacionalmente más económico que los enfoques neuronales.

3.3.3 Rendimiento cuando se utilizan las consultas parafraseadas

```
compute_precision_recall_f1(paraphrased_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 mucho menor debido al desajuste de vocabulario:

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

El rendimiento utilizando búsqueda semántica es mejor, el macropromedio de F1 mejora un 10,8% y su micropromedio un 16%:

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

Por lo tanto, parece que, aunque los *word embeddings* fueron entrenados con materiales muy diferentes de la colección LISA, ayudan a aliviar el problema del desajuste de vocabulario.

3.3.4 Rendimiento cuando se utilizan las consultas en español

```
compute_precision_recall_f1(spanish_run, relevance_judgements_reformat)
```

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

Sin embargo, el rendimiento al enviar consultas en español a una colección de documentos en inglés es bastante bueno y es similar al de las consultas parafraseadas:

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

4 Conclusión

En este *notebook*, hemos explorado el uso de ChromaDB y el *word embedding* multilingüe para realizar una búsqueda semántica en la colección LISA, comparando su rendimiento con la búsqueda léxica basada en BM25 implementada en el anterior *notebook*.

Nuestros experimentos revelaron varias cuestiones importantes relativas a las fortalezas y limitaciones de ambos enfoques.

Si bien la búsqueda semántica con *word embedding* tuvo un rendimiento inferior al de BM25 para las consultas originales (debido a la aplicación del modelo *out-of-domain*), mostró ventajas significativas al manejar consultas parafraseadas y recuperación de información interlingüística.

Para las consultas parafraseadas, la búsqueda semántica demostró su capacidad para mitigar los problemas del desajuste de vocabulario.

En particular, la búsqueda semántica destacó en la recuperación de información interlingüística, logrando un rendimiento comparable en las consultas en español y en las consultas parafraseadas en inglés, mientras que BM25 falló completamente en este escenario.

Estos resultados resaltan el potencial de los enfoques basados en *embeddings* para superar las barreras lingüísticas y los desafíos de reformulación de consultas.

Sin embargo, el rendimiento generalmente inferior respecto al obtenido con BM25 para consultas dentro del dominio subraya la importancia del ajuste de los modelos de *embedding* para los dominios específicos, y la relevancia de los métodos de búsqueda léxica tradicionales.

En el próximo *notebook* exploraremos enfoques híbridos que combinan las fortalezas de la búsqueda léxica y semántica mediante algoritmos de *reranking*.