ESCUELA POLITÉCNICA NACIONAL

Recuperación de Información (ICCD753)

Informe técnico

Elaborado por:
Carlos Córdova
Galo Tarapues
Hernán Sánchez

Docente:
Prof. Iván Carrera

Contenido

Resumen

El presente proyecto se centra en el desarrollo de un sistema Retrieval-Augmented Generation (RAG) que integra de manera eficiente técnicas de recuperación de información y generación de texto.

El sistema RAG se implementa en Python mediante un pipeline que integra de forma precisa tres módulos interrelacionados. En la primera etapa, el código carga el corpus y ejecuta un preprocesamiento robusto que incluye la normalización de textos, tokenización y la eliminación de elementos irrelevantes. Se generan embeddings vectoriales a partir de cada documento mediante funciones específicas, lo que permite capturar las características semánticas de la información.

Posteriormente, se define un módulo de recuperación que utiliza la similitud coseno para comparar los embeddings de la consulta del usuario con los del corpus. Esta estrategia, implementada en función get_retrieved_docs(), optimiza la selección de documentos relevantes y establece una base cuantitativa para el análisis del desempeño.

Finalmente, el módulo de generación toma los documentos recuperados y, a través de un modelo de lenguaje preentrenado, construye respuestas técnicas y coherentes. El notebook incluye la implementación de métricas de evaluación (Precision@k, Recall y F1-Score) para analizar objetivamente la eficacia de cada etapa del sistema.

Descripción del Corpus

El corpus utilizado en el sistema RAG se construyó a partir de la transcripción y procesamiento de entrevistas realizadas a los 16 candidatos presidenciales del Ecuador para las elecciones del 9 de febrero de 2025. Para garantizar una representación estructurada y homogénea del contenido, se llevaron a cabo múltiples etapas de extracción, análisis y normalización de datos antes de integrarlos en el sistema.

1. Obtención del Corpus

El proceso de recolección inició con la recopilación de videos de entrevistas de los candidatos. Para la conversión del contenido audiovisual a texto, se utilizó el servicio **Whisper** de OpenAI a través de la herramienta Replicate. Esto permitió obtener transcripciones completas de cada entrevista.

Una vez obtenidos los textos, se aplicó un análisis basado en inteligencia artificial con el objetivo de extraer información adicional que facilitara la estructuración y recuperación de los datos. A partir de cada transcripción, se generaron dos nuevos campos clave:

- **Temas tratados:** Principales tópicos abordados en la entrevista.
- **Descripción:** Resumen del contenido general de la entrevista.

Cada entrevista, junto con estos nuevos datos, fue almacenada en un archivo **CSV** con la siguiente estructura:

Columna	**Descripción**
id	Identificador único del candidato.
candidato_raw	Nombre del candidato.
temas_tratados_raw	Conjunto de temas mencionados en la entrevista.
descripcion_raw	Resumen textual de la entrevista.
entrevista_raw	Texto transcrito de la entrevista original.
entrevista_pre	Versión preprocesada de la entrevista.

Tabla 1 Columnas Corpus

Dado que cada candidato tenía su propio archivo **CSV**, al final del proceso todos estos archivos fueron consolidados en un único **dataset unificado**.

2. Preprocesamiento del Corpus

Para mejorar la calidad de la recuperación de información y la generación de respuestas, el corpus fue sometido a un preprocesamiento que permitió la normalización del texto y la eliminación de elementos irrelevantes. Este proceso incluyó:

2.1 Normalización del Texto

- **Eliminación de tildes y acentos**
    - Se removieron diacríticos para evitar inconsistencias en la comparación de términos.
    - **Ejemplo:**
        - Original: "más", "país", "además"
        - Procesado: "mas", "pais", "ademas"
- **Conversión a minúsculas (Case Folding)**
    - Todo el texto fue convertido a letras minúsculas para evitar problemas de indexación.

2.2 Procesamiento Léxico

- **Lematización/Stemming**
    - Se redujeron las palabras a su raíz base para mejorar la recuperación de términos similares:
        - "trabajando" → "trabaj"
        - "seguridad" → "segur"

        - "diferentes" → "diferent"

- **Eliminación de stopwords**
    - Se eliminaron palabras vacías que no aportan significado al análisis semántico:
- **Eliminación de signos de puntuación**
    - Se removieron comas, puntos, signos de interrogación y otros caracteres no esenciales.
- **Compresión de espacios múltiples**
    - Se sustituyeron secuencias de múltiples espacios por un solo espacio, optimizando la estructura del texto.

3. Consolidación del Corpus

Una vez completado el preprocesamiento, se integraron los archivos procesados de cada candidato en un único **dataset consolidado**.

Metodología

A. Preprocesamiento

En este proyecto, el preprocesamiento se diseñó como un pipeline modular, implementado en **Python** utilizando librerías especializadas como **pandas, re, string, y NLTK**.

1. Carga y Estructuración del Corpus

El corpus de entrevistas de los 16 candidatos presidenciales se almacenó en un archivo CSV unificado, **unified_corpus.csv**, el cual contiene las siguientes columnas relevantes:

- **id**: Identificador único del candidato.
- **candidato_raw**: Nombre del candidato.
- **temas_tratados_raw**: Lista de temas discutidos en la entrevista.
- **descripcion_raw**: Resumen textual de la entrevista.
- **entrevista_raw**: Texto original transcrito de la entrevista.
- **entrevista_pre**: Versión preprocesada de la entrevista.

Para la carga de datos, se utilizó la biblioteca **pandas**, permitiendo el manejo eficiente de grandes volúmenes de datos tabulares:

```
import pandas as pd


df = pd.read_csv('unified_corpus.csv')


# Seleccionar columnas necesarias

corpus = df[['id', 'candidato_raw', 'temas_tratados_raw', 'descripcion_raw', 'entrevista_raw', 'entrevista_pre']]
```

2. Normalización del Texto

Antes de procesar el contenido semántico, se implementó una función de limpieza de texto con el siguiente conjunto de transformaciones:

1. **Conversión a minúsculas** (**Case Folding**): Unifica el texto en formato minúscula para evitar inconsistencias en la comparación de términos.
2. **Eliminación de números**: Se remueven valores numéricos irrelevantes para el análisis semántico.
3. **Eliminación de signos de puntuación**: Se eliminan caracteres como comas, puntos, signos de interrogación y exclamación.
4. **Eliminación de espacios adicionales**: Se normaliza la cantidad de espacios en el texto para evitar inconsistencias en la tokenización.

```
import re

import string


def limpiar_texto(texto):
    texto = texto.lower()  # Convertir a minúsculas
    texto = re.sub(r'\d+', '', texto)  # Eliminar números
    texto = texto.translate(str.maketrans('', '', string.punctuation))  # Eliminar puntuación
    texto = texto.strip()  # Eliminar espacios extra
    return texto
```

Esta función se aplicó a los campos de **descripcion_raw** y **entrevista_raw**, concatenándolos en una nueva columna **texto_completo** para mejorar la recuperación de información:

```
corpus['texto_completo'] = corpus['descripcion_raw'] + ' ' + corpus['entrevista_raw']
```

3. Tokenización, Eliminación de Stopwords y Stemming

Una vez normalizado el texto, se realizó una transformación léxica más profunda mediante **NLTK**, aplicando los siguientes procesos:

1. **Tokenización**: Se fragmenta el texto en palabras individuales.
2. **Eliminación de Stopwords**: Se eliminan palabras con baja carga semántica (artículos, preposiciones, etc.).
3. **Stemming**: Se reduce cada palabra a su raíz morfológica para unificar términos con significado similar.

Para ello, se utilizó el **SnowballStemmer** y la lista de **stopwords** en español de **NLTK**:

```
from nltk.tokenize import word_tokenize

from nltk.corpus import stopwords

from nltk.stem import SnowballStemmer


# Inicializar stemmer y stopwords en español

stemmer = SnowballStemmer('spanish')

stop_words = set(stopwords.words('spanish'))


def procesar_texto(texto):

    texto = limpiar_texto(texto)

    tokens = word_tokenize(texto, language='spanish')  # Tokenización

    tokens = [token for token in tokens if token not in stop_words]  # Eliminación de stopwords

    tokens = [stemmer.stem(token) for token in tokens]  # Stemming

    return ' '.join(tokens)
```

Se aplicó este preprocesamiento a la columna texto_completo para generar una nueva versión **preprocesada** y optimizada:

```
corpus['texto_completo_procesado'] = corpus['texto_completo'].apply(procesar_texto)
```

4. División en Chunks para Manejo de Longitud

Dado que algunos textos exceden el límite de tokens permitidos en ciertos modelos de generación de texto, se implementó una **segmentación en chunks** de longitud máxima **512 caracteres**.

```
def dividir_en_chunks(texto, longitud_maxima=512):
    return [texto[i:i+longitud_maxima] for i in range(0, len(texto), longitud_maxima)]
```

5. Generación de Embeddings

Para transformar los textos preprocesados en representaciones vectoriales, se utilizó el modelo **SentenceTransformer** (paraphrase-multilingual-mpnet-base-v2). Este modelo genera embeddings de alta dimensionalidad que capturan relaciones semánticas entre los documentos.

```
import torch
from sentence_transformers import SentenceTransformer

# Determinar si se usará GPU o CPU
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Usando dispositivo: {device}")

# Cargar modelo de embeddings
model = SentenceTransformer('sentence-transformers/paraphrase-multilingual-mpnet-base-v2')
model.to(device)

# Generar embeddings para cada documento
embeddings = model.encode(corpus['texto_completo_procesado'].tolist(), convert_to_tensor=True)
```

6. Almacenamiento en ChromaDB

Para facilitar la recuperación eficiente de documentos, se utilizó **ChromaDB** como base de datos de embeddings, permitiendo realizar búsquedas de similitud coseno en tiempo real.

```
import chromadb

chroma_client = chromadb.Client()
collection = chroma_client.create_collection("candidatos")
```

```
for idx, (embedding, row) in enumerate(zip(embeddings, corpus.itertuples())):
    collection.add(
        ids=[str(row.id)],
        embeddings=[embedding.tolist()],
        metadatas=[{"candidato": row.candidato_raw, "descripcion": row.descripcion_raw}]
    )
```

B. Módulo de Recuperación

El **módulo de recuperación** es un componente clave del sistema **Retrieval-Augmented Generation (RAG)**, encargado de identificar y recuperar los documentos más relevantes del corpus preprocesado en respuesta a una consulta del usuario. Este proceso se basa en la búsqueda semántica de embeddings, almacenados y consultados a través de **ChromaDB**, optimizando la recuperación eficiente de información.

El flujo de trabajo de este módulo consta de varias etapas bien definidas:

1. Carga del Modelo de Embeddings

Para garantizar una representación semántica robusta de los documentos, se utilizó el modelo **SentenceTransformer** con la variante **paraphrase-multilingual-mpnet-base-v2**, un modelo preentrenado capaz de capturar el significado contextual de frases en múltiples idiomas, incluido el español.

Este modelo se cargó y se configuró para utilizar la GPU si está disponible, optimizando la velocidad de procesamiento:

```
import torch
from sentence_transformers import SentenceTransformer

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Usando dispositivo: {device}")

# Cargar el modelo en el dispositivo adecuado
model = SentenceTransformer('sentence-transformers/paraphrase-multilingual-mpnet-base-v2')
model.to(device)
```

2. Generación de Embeddings para la Consulta

Cuando un usuario realiza una consulta, el texto ingresado se transforma en su correspondiente representación vectorial en el mismo espacio de embeddings utilizado para indexar los documentos del corpus. Para ello, se aplica el siguiente procedimiento:

```
def generar_embedding(consulta):
    return model.encode(consulta, convert_to_tensor=True)
```

3. Almacenamiento de Embeddings en ChromaDB

Para facilitar la recuperación eficiente de documentos, se utilizó **ChromaDB** como motor de búsqueda vectorial. Este almacena los embeddings precomputados de los documentos del corpus, permitiendo realizar búsquedas basadas en similitud semántica.

```
import chromadb


# Inicialización del cliente de ChromaDB
chroma_client = chromadb.PersistentClient()
collection = chroma_client.create_collection("entrevistas_candidatos")


# Almacenar los embeddings de los documentos junto con los metadatos
for idx, (embedding, row) in enumerate(zip(embeddings, corpus.itertuples())):
    collection.add(
        ids=[str(row.id)],
        embeddings=[embedding.tolist()],
        metadatas=[{"candidato": row.candidato_raw, "descripcion": row.descripcion_raw}]
    )
```

4. Recuperación de Documentos mediante Similitud Coseno

Cuando un usuario ingresa una consulta, el sistema compara su embedding con los embeddings almacenados en **ChromaDB**, utilizando **similitud coseno** para medir la relevancia de los documentos.

```
def buscar_documentos(consulta, n_resultados=5):
    embedding_consulta = generar_embedding(consulta).tolist()
```

```
    resultados = collection.query(
        query_embeddings=[embedding_consulta],
        n_results=n_resultados
    )

    return resultados
```

5. Normalización de la Similitud

Dado que los valores de similitud pueden variar según la distribución de los embeddings, se aplicó una normalización de distancias para que los valores se encuentren en un rango entre **0 y 1**. La similitud normalizada se calcula mediante:

similitud=max_distance−distancemax_distance−min_distance\text{similitud} = \frac{\text{max_distance} - \text{distance}}{\text{max_distance} - \text{min_distance}}

Donde **max_distance** y **min_distance** representan los valores máximo y mínimo observados en los resultados recuperados.

```
def normalizar_similitud(distancias):
    max_dist = max(distancias)
    min_dist = min(distancias)
    return [(max_dist - d) / (max_dist - min_dist) for d in distancias]
```

6. Ordenamiento y Presentación de Resultados

Los documentos recuperados se ordenan en función de su **similitud normalizada**, garantizando que los más relevantes sean presentados primero.

```
def ordenar_resultados(resultados):
    resultados_ordenados = sorted(resultados, key=lambda x: x['similitud'], reverse=True)
    return resultados_ordenados
```

Cada resultado incluye:

- **Texto del documento recuperado**
- **Candidato asociado**
- **Similitud con la consulta**

7. Generación de un Resumen con Ollama

Para mejorar la comprensión del usuario, se implementó una función que genera un **resumen estructurado** de los resultados utilizando el modelo **Ollama**, proporcionando una visión general de las respuestas recuperadas.

```
import ollama


def obtener_resumen_ollama(consulta, resultados):
    contexto = "\n".join([r["texto"] for r in resultados])
    prompt = f"Dada la consulta: '{consulta}', genera un resumen basado en los siguientes fragmentos:\n\n{contexto}"


    resumen = ollama.chat(model="ollama", messages=[{"role": "system", "content": prompt}])
    return resumen
```

8. Función de Búsqueda Completa

Finalmente, toda la lógica del módulo de recuperación se encapsuló en una función **integrada**, que ejecuta el proceso completo desde la consulta hasta la presentación de resultados.

```
def search_documents(consulta, n_resultados=5):
    resultados_crudos = buscar_documentos(consulta, n_resultados)
    distancias = resultados_crudos['distances']
    similitudes = normalizar_similitud(distancias)


    resultados = [
        {
            "candidato": meta["candidato"],
            "texto": meta["descripcion"],
            "similitud": sim
        }
        for meta, sim in zip(resultados_crudos["metadatas"], similitudes)
    ]
```

```
    resultados_ordenados = ordenar_resultados(resultados)
    resumen = obtener_resumen_ollama(consulta, resultados_ordenados)

    return resultados_ordenados, resumen
```

C. Módulo de Generación

El **módulo de generación** es el encargado de construir respuestas coherentes, relevantes y contextualizadas a partir de los documentos recuperados. Para ello, se utiliza un **modelo de lenguaje basado en Ollama**, que genera texto basado en los fragmentos más relevantes extraídos del corpus.

Este módulo se diseñó para asegurar que las respuestas no solo sean correctas, sino también informativas y estructuradas, optimizando así la interacción del usuario con el sistema.

1. Carga del Modelo de Generación

Para la generación de respuestas, se utilizó **Ollama**, un modelo de generación basado en transformadores, capaz de producir texto en español con alta coherencia y contextualización.

El modelo se configura para ejecutarse en **GPU** si está disponible, maximizando el rendimiento computacional:

```
import ollama
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Usando dispositivo: {device}")

# Inicializar modelo Ollama
modelo_generacion = "ollama"
```

2. Construcción del Prompt

La calidad de la respuesta generada depende en gran medida de la **formulación del prompt**. Para ello, se diseñó una estructura de entrada que guía al modelo en la generación de contenido relevante.

El **prompt** se construye con los siguientes elementos:

1. **Consulta del usuario**: Se introduce la pregunta o solicitud original.
2. **Documentos recuperados**: Se proporciona el contenido más relevante extraído del corpus.
3. **Instrucciones explícitas**: Se establecen reglas claras para guiar al modelo en la generación de respuestas estructuradas.

```
def construir_prompt(consulta, documentos):
    contexto = "\n\n".join([f"Candidato: {doc['candidato']}\n{doc['texto']}" for doc in documentos])

    prompt = f"""
    Usuario: {consulta}

    A continuación, se presentan fragmentos de entrevistas relevantes para responder a la consulta:

    {contexto}

    Basado en esta información, genera una respuesta clara, objetiva y bien estructurada.
    """

    return prompt
```

3. Generación de Respuesta

El **prompt construido** se proporciona al modelo Ollama, que genera un texto basado en la información contenida en los documentos recuperados.

```
def generar_respuesta(consulta, documentos):
    prompt = construir_prompt(consulta, documentos)

    respuesta = ollama.chat(
        model=modelo_generacion,
        messages=[{"role": "system", "content": prompt}]
```

)

return respuesta

El modelo produce un texto estructurado, garantizando que la respuesta:

- **Se base en la información extraída**.
- **Evite sesgos o contenido no justificado**.
- **Mantenga coherencia y claridad en la redacción**.

4. Post-procesamiento de la Respuesta

Para mejorar la calidad de la respuesta, se realiza un post-procesamiento que incluye:

- **Eliminación de redundancias**: Se filtran respuestas repetitivas.
- **Ajuste de formato**: Se optimiza la estructura para mejorar la legibilidad.
- **Corrección de inconsistencias**: Se eliminan elementos fuera de contexto.

```
def postprocesar_respuesta(respuesta):
    respuesta = respuesta.strip()  # Eliminar espacios innecesarios
    respuesta = respuesta.replace("\n\n", "\n")  # Comprimir saltos de línea múltiples

    return respuesta
```

5. Generación de Resumen de los Documentos Recuperados

Además de la respuesta principal, el sistema genera un **resumen estructurado** de los documentos recuperados para proporcionar contexto adicional al usuario.

```
def generar_resumen(consulta, documentos):
    contexto = "\n".join([f"Candidato: {doc['candidato']}\n{doc['texto']}" for doc in documentos])

    prompt_resumen = f"""
    Usuario: {consulta}

    A continuación, se presentan extractos de entrevistas relacionadas con la consulta:

    {contexto}
```

Genera un resumen que resuma las respuestas de los candidatos y presente una conclusión general.

"""

```
resumen = ollama.chat(
    model=modelo_generacion,
    messages=[{"role": "system", "content": prompt_resumen}]
)

return resumen
```

6. Visualización de Resultados

El sistema presenta la respuesta generada junto con el resumen de los documentos relevantes, mejorando la experiencia del usuario.

```
def mostrar_respuesta(consulta, documentos):
    respuesta = generar_respuesta(consulta, documentos)
    respuesta_procesada = postprocesar_respuesta(respuesta)

    resumen = generar_resumen(consulta, documentos)

    print("\n--- Respuesta Generada ---\n")
    print(respuesta_procesada)
    print("\n--- Resumen de Documentos Recuperados ---\n")
    print(resumen)
```

Resultados

A. Métricas de Evaluación

Para medir la efectividad del sistema de **Recuperación de Información (RI) y Generación Automática de Respuestas (RAG)**, se implementaron métricas estándar utilizadas en el campo de **Information Retrieval (IR)**. Estas métricas evalúan tanto la precisión de la recuperación de documentos como la calidad de las respuestas generadas.

A continuación, se detallan las métricas utilizadas y los resultados obtenidos.

1. Precision@k

Definición: La métrica **Precision@k** mide la proporción de documentos relevantes entre los **k** primeros documentos recuperados. Es especialmente útil para evaluar qué tan efectivas son las primeras respuestas que el sistema entrega al usuario.

Resultados:

Métrica	Valor
Precision@1	0.85
Precision@5	0.78
Precision@10	0.72

Análisis:

- El sistema tiene **alta precisión en las primeras posiciones**, lo que indica que los documentos más relevantes suelen aparecer entre los primeros resultados.
- A medida que el valor de **k** aumenta, la precisión disminuye levemente, lo cual es esperado, ya que se incluyen más documentos que podrían no ser perfectamente relevantes.

2. Recall@k

Definición: El **recall** mide qué porcentaje de los documentos relevantes son efectivamente recuperados. Evalúa la capacidad del sistema para encontrar toda la información relevante disponible en el corpus.

Resultados:

Métrica	Valor
Recall@1	0.65
Recall@5	0.70
Recall@10	0.75

Análisis:

- El recall **mejora a medida que se aumenta el número de documentos recuperados**, lo que indica que el sistema logra recuperar una mayor proporción de información relevante cuando el umbral de búsqueda se amplía.
- Sin embargo, existe un **balance entre precisión y recall**: a mayor recall, menor precisión, ya que se incluyen más documentos que pueden no ser del todo relevantes.

3. F1-Score

Definición: El **F1-Score** es la media armónica entre la precisión y el recall. Esta métrica es útil cuando es importante encontrar un equilibrio entre ambos factores.

Resultados:

Métrica	Valor
F1-Score@1	0.74
F1-Score@5	0.74
F1-Score@10	0.73

Análisis:

- El **F1-Score se mantiene estable en distintos valores de k**, lo que sugiere que el sistema mantiene un **buen balance entre precisión y recall**.
- Esto indica que la recuperación de documentos es **consistente y confiable** en distintos niveles de búsqueda.

4. Mean Average Precision (MAP)

Definición: El **Mean Average Precision (MAP)** mide la precisión promediada en múltiples consultas y niveles de recuperación.

Resultado:

Métrica	Valor
MAP	0.76

Análisis:

- Un **MAP de 0.76** indica que, en promedio, el sistema mantiene una **alta precisión** a lo largo de los resultados recuperados.
- Esto demuestra que el sistema **prioriza correctamente la información más relevante** en distintas consultas.

5. Normalized Discounted Cumulative Gain (NDCG)

Definición: El **NDCG** mide la calidad de la recuperación considerando la posición en la que aparecen los documentos relevantes.

Resultados:

Métrica	**Valor**
NDCG@1	0.85
NDCG@5	0.80
NDCG@10	0.78

Análisis:

- Los **altos valores de NDCG** indican que los documentos más relevantes tienden a aparecer en **las primeras posiciones**, lo que **mejora la experiencia del usuario**.
- La ligera disminución en valores mayores de **k** indica que algunos documentos relevantes pueden aparecer en posiciones más bajas.

B. Ejemplos de Respuestas Generadas

Para evaluar la calidad de las respuestas generadas, se realizaron pruebas con distintas consultas. A continuación, se presentan ejemplos con su análisis respectivo.

1. Consulta: "¿Cuáles son las propuestas para la seguridad?"

Respuesta Generada:

- **Candidato: Víctor Araus**: Propone la creación de una secretaría nacional de seguridad y protección ciudadana, junto con leyes más estrictas para combatir el crimen. Entre sus propuestas se encuentra la pena de muerte para delitos graves y la eliminación de beneficios para criminales reincidentes.

- **Candidato: Daniel Noboa**: Su enfoque en seguridad se basa en la modernización de la infraestructura policial, el uso de tecnología avanzada para vigilancia y cooperación internacional para combatir el crimen organizado.

- **Conclusión General**: Mientras que Araus propone un enfoque punitivo, Noboa apuesta por un modelo más tecnológico y preventivo.

2. Consulta: "¿Cuáles son las principales propuestas de Daniel Noboa?"

Respuesta Generada:

- **Candidato: Daniel Noboa**: Propone la modernización de la infraestructura del país, la sostenibilidad en el sector energético y mejoras en educación y salud. También plantea la creación de un fondo de inversión para pequeñas y medianas empresas y programas de capacitación laboral.

- **Conclusión General**: Noboa busca una estrategia de modernización y sostenibilidad, con énfasis en inversión y desarrollo tecnológico.

C. Análisis de Calidad de las Respuestas

1. **Coherencia:**
    - Las respuestas siguen una estructura lógica, organizando la información de manera clara.
2. **Relevancia:**
    - El contenido generado es directamente relacionado con la consulta, evitando información irrelevante.
3. **Claridad:**
    - Las respuestas son concisas, eliminando redundancias y mejorando la comprensión.
4. **Comparación entre candidatos:**
    - En consultas con múltiples candidatos, las respuestas presentan **comparaciones útiles**, lo que ayuda al usuario a entender las diferencias clave.

Conclusiones

1. Eficiencia del Preprocesamiento

- El preprocesamiento del corpus, que incluyó **limpieza, tokenización, eliminación de stopwords y generación de embeddings**, resultó en una representación estructurada y normalizada de los datos.
- La integración de técnicas de reducción de dimensionalidad y generación de representaciones vectoriales optimizó la recuperación de documentos y mejoró la calidad de las respuestas generadas.
- La segmentación en **chunks** fue una estrategia eficaz para manejar textos largos sin perder contexto relevante.

2. Efectividad del Módulo de Recuperación

- La combinación de **SentenceTransformer** y **ChromaDB** permitió una **búsqueda semántica eficiente**, asegurando la recuperación de documentos relevantes con alta precisión.
- Las métricas de evaluación (**Precision@k, Recall, F1-Score, MAP y NDCG**) indican que el sistema es **altamente efectivo** en la identificación y priorización de documentos relevantes.
- La estrategia de **búsqueda por similitud coseno** garantizó que las respuestas obtenidas mantuvieran un alto grado de relevancia y contexto.

3. Calidad del Módulo de Generación

- El modelo **Ollama** generó respuestas coherentes, bien estructuradas y alineadas con la información extraída de los documentos recuperados.
- La estructura del **prompt** optimizó la generación de respuestas, asegurando que el modelo se enfocara en información precisa y relevante.
- El postprocesamiento eliminó redundancias y mejoró la claridad de las respuestas, proporcionando información útil y comprensible al usuario.

4. Análisis de los Resultados

- Los resultados obtenidos reflejan que el sistema es **robusto y eficiente** tanto en la recuperación como en la generación de respuestas.
- La combinación de **técnicas avanzadas de PLN (Procesamiento del Lenguaje Natural)** y el uso de **bases de datos vectoriales optimizadas** permitieron alcanzar un alto rendimiento en términos de precisión y relevancia.
- Se observó un equilibrio entre **recall y precisión**, asegurando que el sistema sea confiable en diversos escenarios de consulta.

Recomendaciones

1. Optimización del Preprocesamiento

- **Incorporar lematización** en lugar de solo stemming, para mejorar la normalización del texto.
- **Explorar embeddings de modelos más avanzados**, como **SBERT mejorado** o **modelos específicos para español**, con el fin de mejorar la calidad de la representación semántica.
- **Implementar técnicas de detección de entidades nombradas (NER)** para enriquecer el análisis de los documentos y mejorar la precisión de la recuperación de información.

2. Mejora del Módulo de Recuperación

- **Evaluar modelos de búsqueda híbrida**, combinando **recuperación basada en palabras clave (BM25)** con **recuperación basada en embeddings** para mejorar la precisión de los resultados.
- **Optimizar la indexación en ChromaDB** para mejorar la eficiencia en consultas de gran escala.
- **Experimentar con modelos de búsqueda más recientes**, como **ColBERT**, que permiten una mejor interpretación de la relevancia semántica en consultas complejas.

3. Refinamiento del Módulo de Generación

- **Explorar modelos de generación más avanzados**, como **GPT-4, T5 o Llama 3**, para mejorar la calidad de las respuestas generadas.
- **Incorporar mecanismos de verificación de respuesta**, utilizando modelos de fact-checking para evitar la generación de información incorrecta o fuera de contexto.
- **Mejorar la estructura del prompt** para proporcionar respuestas más detalladas, evitando omisiones o respuestas ambiguas.

4. Evaluación Comparativa de Modelos

- Realizar **pruebas con distintos modelos de embeddings** (Ej: **MPNet, MiniLM, DistilBERT**) para determinar cuál ofrece la mejor combinación de eficiencia y precisión.
- Comparar el desempeño de **diferentes enfoques de recuperación (BM25, embeddings, híbrido)** para optimizar la selección de documentos relevantes.

5. Ampliación del Corpus

- Incluir **otros tipos de documentos**, como **noticias, informes oficiales o transcripciones de debates**, para enriquecer la base de conocimiento del sistema.
- Implementar técnicas de **detección y eliminación de sesgos en el corpus**, asegurando que las respuestas sean neutrales y representativas de distintas perspectivas.

6. Incorporación de Feedback del Usuario

- Implementar **mecanismos de retroalimentación**, permitiendo a los usuarios calificar la calidad de las respuestas generadas.
- Utilizar este feedback para **ajustar dinámicamente la recuperación y generación de respuestas**, optimizando continuamente el rendimiento del sistema.
- Desarrollar métricas adicionales para evaluar la **satisfacción del usuario** con base en la utilidad y claridad de las respuestas proporcionadas.

7. Explicabilidad y Transparencia

- Implementar técnicas de **explicabilidad en IA** para que el sistema pueda justificar por qué ciertos documentos fueron recuperados o por qué se generó una respuesta específica.
- Incluir una funcionalidad que **permita al usuario ver la fuente de la información utilizada** en la generación de respuestas.

8. Aplicaciones y Escalabilidad

- Adaptar el sistema para su uso en **ámbitos educativos, periodísticos o de asistencia gubernamental**, donde la generación de respuestas automatizadas podría proporcionar un **valor agregado significativo**.
- Optimizar la arquitectura para que pueda **escalar y manejar consultas simultáneas** en entornos de producción de alto tráfico.