Tecnicatura Universitaria en Inteligencia Artificial

Universidad Nacional de Rosario

Procesamiento del Lenguaje Natural

2024

Alumno: Lucas Gauto

Trabajo práctico final

Ejercicio 1

Introducción

En el campo del Procesamiento de Lenguaje Natural (NLP), la generación automática de texto ha evolucionado significativamente con la introducción de modelos como el Retrieval Augmented Generation (RAG). Este modelo combina técnicas de recuperación y generación de texto para producir resultados que son más coherentes y relevantes con respecto al contexto específico.

La idea base consiste en dotar a un LLM de diversas fuentes de información específicas sobre un tema en concreto para que esta pueda utilizarlas a la hora de realizar respuestas sobre dichos temas.

En este trabajo práctico en concreto, la información proporcionada al modelo es sobre una popular serie de anime llamada Neon Genesis Evangelion, dividiendo las fuentes en tres:

- Documentos de texto: los cuales poseen un resumen sobre cada capítulo e información sobre los antagonistas de la serie;
- Dataframe de pandas: que proporciona datos sobre las películas de la saga, incluyendo aspectos como director, recaudación y nombre.
- Base de datos de grafos: que contiene datos sobre los personajes de la serie.
- Desarrollo detallado de los pasos para la solución del problema, y justificaciones correspondientes.

Para la implementación del chatbot con RAG se debieron tener en cuenta varias partes del Natural Lenguage Processing.

Web scrapping

Para comenzar, una de las mayores dificultades consistió en la recolección de la información y su adecuación para que esta esté estandarizada y así lograr una mayor efectividad en las respuestas.

Para este punto se utilizó el web scrapping aprendido a lo largo del cuatrimestre y se realizó una limpieza y formateo distinto a los textos obtenidos según cuál sería su destino (dataframe, texto o grafos).

Organización del proceso RAG

Una de las partes del trabajo práctico que más dificultades me generó era lograr comprender correctamente cómo es el flujo en una consulta a un modelo implementado con RAG.

Comprender que era posible clasificar la consulta en varias etapas y distintas categorías me ayudó a ver que, de esa manera, podía lograrse una mejor precisión en las respuestas. Esto se ve plasmado principalmente cuando se consulta a la base de datos de grafos, siendo el flujo el siguiente:

Consulta -> Se clasifica en personajes (bbdd de grafos) -> Se clasifica en un personaje en concreto dependiendo de cuál se nombre -> Se consulta la información de dicho personaje.

Aspectos técnicos del trabajo práctico

Sin dudas, otros aspectos tuvieron sus dificultades y problemáticas, las cuales se fueron solucionando de manera empírica en la mayoría de los casos.

Me refiero principalmente a temas como pueden ser el tamaño de los chunks en los que se dividen los textos (tener en cuenta el límite de tokens del LLM), o la elección del método para realizar las clasificaciones (utilizando modificaciones del prompot en un LLM configurado para clasificar o valiéndome de técnicas más tradicionales de clasificación dadas en clase).

Código en detalle

1. **Obtención de datos**
    1. **Capítulos:** Se utilizó BeautifulSoup y Request para realizar el scrapping de los distintos capítulos de la serie. Los resúmenes fueron conseguidos de la página https://evangelion.fandom.com/es/wiki, donde se identificó que la url cambiaba en el último tramo según el episodio que se estuviera visualizando (https://evangelion.fandom.com/es/wiki/EP01, para el episodio 1, etc.) y, sabiendo que la serie tiene 27 capítulos, se extrajo archivos txt para cada capítulo. Estos archivos fueron guardados en la carpeta Capítulos y posteriormente fueron formateados para quitar los saltos de línea doble.
    2. **Personajes:** nuevamente hice uso de la misma página donde se identificó que la url cambiaba según el personaje del cual se estuviera dando información, de forma que construí manualmente una lista de personajes y realicé el mismo procedimiento que con los capítulos. La sección personajes fue especialmente complicada ya que se requería que su formato se adaptara para luego utilizar

esta información para construir una base de datos de grafos; por este motivo se realizó una limpieza más exhaustiva, formateando el texto con la función `informaciónPersonaje` la cual a su vez hace uso de `reemplazar_nuevas_lineas`. La explicación de ambas a continuación:

1. `reemplazar_nuevas_lineas(texto)`: reemplaza múltiples saltos de línea por uno (\n\n) solo utilizando la expresión regular `r'\\n{2,}'` que busca dos o más caracteres de salto de línea **consecutivos** en el texto. Por otro lado, también se asegura de que si hay un sólo salto de línea entre dos bloques de texto, éste no se incluye en el reemplazo. De esta forma logré aislar la parte superior del texto (que tenía información relevante para almacenar en formato de diccionario) del cuerpo.
2. `informaciónPersonaje(path)`: esta función hace uso de la anterior y separa por salto de línea (\n\n) para que quede cada bloque de texto en una lista distinta. Posteriormente se comprueba que no se trate del cuerpo del texto (longitud menor a 1000) y se inserta esta información en un diccionario. Para finalizar se agrega la key 'Resumen' al diccionario, la cual contiene el cuerpo del documento.

Con el uso de estas dos funciones podemos pasar de un texto en este formato:

```
Kaworu Nagisa
渚 カヲル

                                                    Normal



                                                    Plug
Suit



Afiliación
NERV  SEELE  Ángeles
```

Rango
Fifth Children

Edad
15 años

Sexo
Masculino

Primera aparición
El último mensajero [NGE] Desaparecido [Manga]
Evangelion: 1.0 [RB]

Seiyū
Akira Ishida

Doblador Latino
Alberto Bernal (EoE, Death and Rebirth y tercer doblaje de NGE) Federico Llambí (Evangelion: 3.0+1.0, segundo doblaje de Evangelion: 1.0, 2.0 y 3.0) Ernesto Lezama (primer doblaje de NGE, primer doblaje de Evangelion: 1.0, 2.0 y 3.0) Edson Matus (segundo doblaje de NGE)

Doblador en España
Jordi Pons

Kaworu Nagisa (渚 カヲル, , Nagisa Kaworu ?) es el Fi(etc...)

A un diccionario en este formato:

```
{'Kaworu Nagisa': '渚 カヲル',
 'Afiliación': 'NERV  SEELE  Ángeles',
 'Rango': 'Fifth Children',
 'Edad': '15 años',
 'Sexo': 'Masculino',
```

'Primera aparición': 'El último mensajero [NGE] Desaparecido [Manga] Evangelion: 1.0 [RB]',
'Seiyū': 'Akira Ishida',
'Doblador Latino': 'Alberto Bernal (EoE, Death and Rebirth y tercer doblaje de NGE) Federico Llambí (Evangelion: 3.0+1.0, segundo doblaje de Evangelion: 1.0, 2.0 y 3.0) Ernesto Lezama (primer doblaje de NGE, primer doblaje de Evangelion: 1.0, 2.0 y 3.0) Edson Matus (segundo doblaje de NGE)',
'Resumen': 'Doblador en España\\nJordi Pons\\nKaworu Nagisa (渚 カヲル, , Nagisa Kaworu ?) es el Fifth ...
(etc)
}

3. **Ángeles:** con los ángeles se utilizó el mismo método para la extracción de los datos. Además, se sumó un documento de información general sobre los ángeles.
4. **Películas:** en el caso de las películas, la información respectiva a ellas se extrajo de las distintas páginas que tiene *wikipedia* sobre las películas de la saga. Concretamente se extrajo las tablas de información de cada página para luego realizar un dataframe de pandas. Tareas de limpieza fueron realizadas para tener los datos en un formato homogéneo.

2. **Adecuación de los datos**
    1. **Datos tabulares:** como se mencionó posteriormente, los datos tabulares consisten en la tabla de las películas.
    2. **Base de datos de grafos:** para la base de datos de grafos se utilizó la librería de nodos *networkx*. Con el diccionario creado anteriormente se crearon los nodos, y las relaciones se basaron en la familia de cada personaje. Para las relaciones se utilizó la función `obtener_familia`, la cual identifica los miembros de la familia de cada personaje y los separa para establecer las relaciones respectivas.
        1. `obtener_familia(personaje:str)`: esta función obtiene el atributo ‘Familia’ de cada diccionario de personajes y los separa en una lista por el delimitador ‘,’. Posteriormente extrae la relación (la cual está entre paréntesis). Se verifica si existe el nodo del familiar y, si lo hace, se crea la relación entre los nodos; si no existe el nodo, se crea.

Por ejemplo, para el personaje ‘Misato Katsuragi’, los familiares se encuentran en este texto: 'Dr. Katsuragi (padre fallecido) Pen Pen (mascota) Tutora de Shinji Ikari y Asuka Langley Soryu'.

3. `{'Misato Katsuragi': '葛城 ミサト',`
4. `'Edad': '29 años  32 años (Evangelion ANIMA)  43 años (3.0)',`
5. `'Sexo': 'Femenino',`
6. `'Familia': 'Dr. Katsuragi (padre fallecido)  Pen Pen (mascota)  Tutora de Shinji Ikari y Asuka Langley Soryu',`
7. `...}`
8. Utilizando la función, logramos obtener un resultado como este:
9. `Relación: Misato Katsuragi - Dr. Katsuragi, Tipo: padre fallecido`
10. Adicionalmente se añadió información extraída manualmente a los nodos.

3. **Preparación de funciones necesarias para la ejecución final**
    1. **Funciones para embeddings**: para realizar los embeddings se crrearon algunas funciones:
        1. `cargarModeloEmbeddings()`: como su nombre lo indica, carga el modelo, utilizando `'distiluse-base-multilingual-cased-v1'` BERT.
        2. `obtenerEmbeddings(oraciones:list, model_embed)`: recibe una lista de oraciones o frases de un texto y, con la utilización de un modelo de embeddings genera los embeddings necesarios.
        3. `contextoMasSimilar(embeddings, modelo, query:str, tpn:int = 1)`: obtiene las topn embeddings más similares utilizando una comparación de coseno de la query y los embeddings. Esta función fue creada pero posteriormente se decidió utilizar otros métodos para obtener el contexto más similar.
        4. `generarChunks(contenido_del_texto_o_directorio:str, directorio = True, tamañoChunk:int=500, chunkOverlap:int=20)->list` : si se le pasa un texto, lo

divide en sus respectivos chunks. Si se le pasa un directorio, lee los documentos en él, los junta y los divide en chunks.

2. **Base de datos vectorial:** en este punto utilicé chormadb como base de datos vectorial para almacenar los embeddings, pero antes de eso tuve que realizar algunos pasos previos:
    1. **Agregar los chunks de los capítulos a la base de datos vectorial:** utilizando la función previamente definida `generarChunks` se obtuvo los chunks respectivos a los textos contenidos en el directorio "capítulos". El tamaño de los chunks fue decidido de manera empírica, se encontró que a tamaños muy pequeños el modelo obtenía contextos no tan relacionados con la consulta debido a su similitud con las palabras. Por último, se anexó cada chunk con sus embeddings y su categoría a la base de datos vectorial.
    2. **Agregar los chunks de los angeles a la base de datos vectorial:** al igual que el paso anterior, se anexaron los chunks del directorio "angeles" a la base de datos vectorial.
    3. **Agregar los chunks del dataframe de películas a la base de datos vecotrial:** por último, se realizó el mismo proceso con el dataframe buscando captar el contenido semántico del mismo dentro de la base de datos. Se tomó cada fila como un texto y se obtuvo los embeddings.
    4. **Realizar consulta:** se definió una función definitiva para obtener los contextos más similares en mi base de datos vectorial. La función es `obtenerContextosMasSimilares(query: str, topn: int = 3)` que toma una consulta, el número de contextos más similares buscado y devuelve un diccionario con información respectiva a cada contexto, como la categoría y la distancia de la consulta actual.
3. **Preparado del modelo:** para preparar el modelo se utilizaron las funciones dadas en clases junto con la api al modelo zephyr 7b beta de hugging face: https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7b-beta.
    1. `zephyr_chat_template`: renderiza la plantilla con los mensajes proporcionados.

2. `generate_answer(prompt: str, max_new_tokens: int = 768)` : hace la llamada a la api y obtiene una respuesta para el prompt proporcionado.
3. `prepare_prompt(query_str: str, context_str: str)` : crea el prompt a pasarle a la función `generate_answer` a partir de un contexto y la consulta proporcionada.
4. `responder(query:str, contexto:str)`: engloba las últimas funciones, prepara el prompt y responde la consulta dado un contexto obtenido.

4. **Clasificador basado en regresión logística:** para la clasificación basada en regresión logística se definió varias consultas genéricas de cada categoría y se entrenó un modelo de clasificación múltiple. Por último, se definió una función que, dada una consulta, obtiene su clasificación en una de las tres categorías: "capítulos o ángeles", "Películas" o "Personajes". La función de clasificación definida se llama `clasificadorRegresionLogistica`.
5. **Clasificador basado en LLM:** de forma similar a lo observado en la preparación del modelo, se definieron las mismas funciones, pero esta vez para realizar una clasificación. El mayor cambio lo encontramos en `prepare_promtClf`, ya que se definió un prompt específicamente orientado a la clasificación:

6. `"Clasifica la siguiente pregunta en una de estas tres categorías: 'Películas', 'Capítulos o ángeles' o 'Personajes', entendiendo que se refiere a la serie de anime llamada Neon Genesis Evangelion"`
7. `"donde la pregunta puede ser sobre los personajes, sobre las películas (recaudación, quién la dirigió, etc.) o sobre información realcionada a los capítulos de la serie o a las criaturas llamadas ángeles. Sólo responderás la clasificación deseada, sin más texto adicional."`
8. `"Ejemplo: ¿Quién recoge a Shinji cuando llega a tokyo? - Capítulos o ángeles"`
9. `"¿Qué ángel ataca en el primer capítulo? - Capítulos o ángeles"`
10. `"You can (not) advance - Películas"`
11. `"You can (not) re do - Películas"`

```
12.   "You are (not) alone - Películas"
13.   "¿Quién es Shinji Ikari? - Personajes"
14.   "Pregunta: {query_str}\\n"
15.   "Respuesta: "
```

Por último se definió la función que realizaría esta clasificación: `clasificadorLLM`.

16. **Modelo para responder consultas sobre el dataframe:** tras comprobar los resultados del clasificador basado en una LLM, quise utilizar el mismo enforque para responder consultas relacionadas al dataframe. Para ello, nuevamente retoqué la función encargada de preparar los prompts, ahora llamada `prepare_promptPeliculas`, dándole la siguiente definición:

```
17.   "Responda la consulta teniendo en cuenta que el
      contexto proporcionado es está en formato csv
      (separado por comas) y cuyas columnas son:"
18.   "'Título', 'Dirección', 'Dirección artística',
      'Producción', 'Guion','Basada en', 'Música',
      'Sonido', 'Fotografía', 'Montaje','Protagonistas',
      'País', 'Año', 'Estreno', 'Género',
      'Duración','Clasificación', 'Idioma(s)',
      'Productora', 'Distribución','Recaudación',
      'Estudio', 'Actores de voz', 'Divisa'"
19.   "Contexto: {context_str}"
20.   "Pregunta: {query_str}\\n"
21.   "Respuesta: "
```

22. **Modelo para responder consultas sobre la base de datos de grafos:** en el caso de los grafos, pensé que sería una mejor alternativa utilizar una regresión logística para “categorizar” los distintos personajes. La idea era que cada personaje fuera una categoría distinta y se clasificara la consulta en estos para luego acceder a su información detallada. En un principio intenté realizar este mismo enfoque pero basado en un LLM, pero encontré con que para esta tarea requería más precisión en las respuestas, y opté por algo más tradicional. El resultado, si bien lejos de ser perfecto, fue un clasificador que me permitía obtener respuestas lo suficientemente coherentes sobre el personaje consultado. La función definida para la clasificación de los personajes es `clasificadorDePersonaje`.

4. **Main:** por último, en la sección main se integró todo para lograr correr un pequeño programa de consola donde el usuario puede realizar preguntas y el LLM responde. Para empezar la consulta se clasifica según tres categorías: Películas, Personajes o Capítulos o ángeles. Si la consulta fue sobre las películas, se consultará al dataframe en busca de un contexto y se obtendrá una respuesta. Si la consulta fue sobre los personajes, se utilizará `clasificadorDePersonajes` para obtener la información respectiva al personaje consultado. De la misma manera, si la consulta fue sobre los ángeles o capítulos, se responderá realizando una búsqueda en la base de datos vectorial.

Conclusiones

Gracias a este trabajo práctico me vi obligado a pensar en muchas alternativas para que el sistema funcionara de la forma correcta. Si bien (e incluso con la extensión de los plazos) me he encontrado con falta de tiempo para probar otras ideas que se me iban ocurriendo, creo que el resultado es mucho mejor del que esperaba ser capaz de producir.

Tuve que incorporar conceptos clave de la materia como las distintas formas de tokenización, búsqueda por similitud, chunks y sus tamaños y overlap, así también como la utilización de otras herramientas aprendidas en otras materias como la regresión logística.

Termino el trabajo práctico con una idea más certera de cómo podría funcionar la implementación de un chatbot utilizando información personal, así como con muchas ideas de implementaciones en proyectos propios.

Enlaces a los modelos y librerías utilizados

Las librerías utilizadas fueron las siguientes:

- Networx: https://networkx.org/
- Numpy: https://numpy.org/
- Requests: https://requests.readthedocs.io/en/latest/
- BeautifulSoup: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- Langchain: https://python.langchain.com/v0.2/docs/introduction/
- SentenceTransofrmer: https://sbert.net/

- Llama_index: https://docs.llamaindex.ai/en/stable/
- Chromadb: https://docs.trychroma.com/

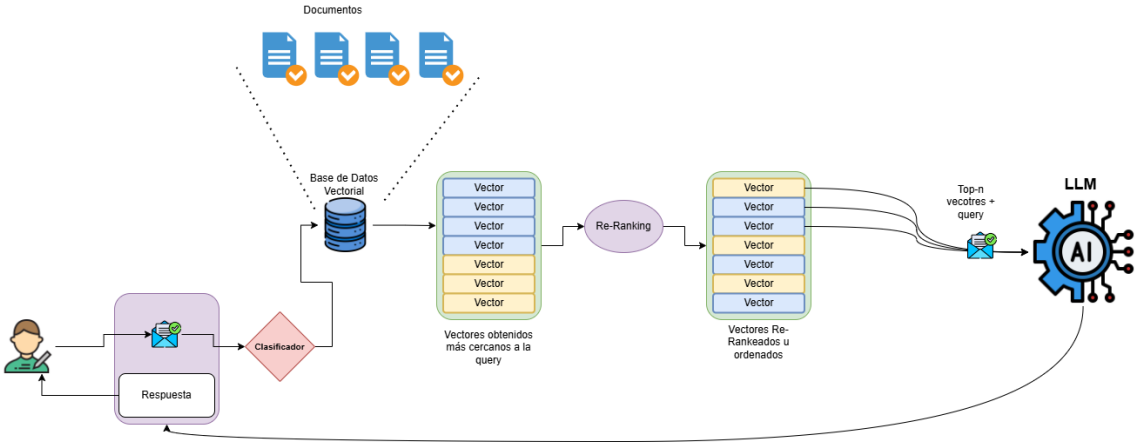
Ejercicio 2:

Desarrollo de las preguntas

Explique con sus palabras el concepto de Rerank y como impactaría en el desempeño de la aplicación. Incluir un diagrama de elaboración propia en la explicación.

El proceso de re-ranking consiste en, una vez seleccionados aquellos embeddings más similares a la consulta realizada, realizar un ordenamiento de los mismos basado en una puntuación de coincidencia. Es una forma de asegurarse que el LLM está recibiendo el contexto adecuado y no simplemente los primeros obtenidos al buscar por similitud con la consulta.

Recordemos que los contextos obtenidos por similitud de coseno con la consulta no siempre serán los que respondan a esta. La razón es que, al dividir un documento en distintos chunks con tamaños menores, se puede escapar de los mismos información necesaria para una consulta específica. Por ello, utilizar re-ranking luego de obtener los contextos más similares podría aumentar la precisión del LLM a la hora de responder.



¿En qué sección de su código lo aplicaría?

Actualmente en mi código solo obtengo el primer contexto devuelto tras hacer la comparación con la query, por lo que previamente debería modificarlo para obtener varios contextos y, posteriormente, aplicar el re-ranking para obtener los tres o cuatro mejores. Por lo tanto, el re-ranking lo aplicaría luego de obtener los contextos más similares con

`obtenerContextosMasSimilares`: almacenaría los más similares en alguna estructura de datos como una lista y posteriormente los re ordenaría con un re-ranking para quedarme con los mejores.

Detalle de las fuentes de la información utilizadas y sus autores.

- Improve Retrieval Augmented Generation (RAG) with Re-ranking | by ASHPAK MULANI | Medium, de Ashpak Mulani, febrero 24 de 2024.