UNIVERSIDAD NACIONAL DE ROSARIO (UNR)

FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA (FCEIA)

TECNICATURA UNIVERSITARIA EN INTELIGENCIA ARTIFICIAL (TUIA)

Docentes de la cátedra:

Manson, Juan Pablo

Gery, Alan

Trabajo Práctico 2 (TP2)

Estudiantes:

Francisco J. Alomar

AÑO 2024 (1er CUATRIMESTRE), Rosario.

INTRODUCCIÓN

En el siguiente informe describimos el proceso de elaboración de un *chatbot* experto en la temática deuda externa argentina mediante *Retrieval Augmented Generation* (RAG). El *Large Language Model* (LLM) que hemos utilizado es zephyr-7b-beta, disponible en Hugging Face. Para ello, se consultan tres bases de datos para contextualizar al modelo respecto de la pregunta que haga un usuario y de acuerdo a un clasificador. Si la pregunta es sobre la historia de la deuda externa, el código dará como contexto elementos de una base de datos vectorial, indexada, alojada en Chroma DB, que es generada a partir de un conjunto de artículos y textos en pdf sobre la deuda externa. Si la consulta es sobre el monto de la deuda para un año dado, los datos correspondientes surgirán de una base de tabular (.csv), un archivo confeccionado por nosotros con datos disponibles de distintas fuentes. Si la consulta es sobre alguna institución internacional de crédito (FMI, BID, Banco Mundial o Club de París), la información se construirá a partir de la consulta a Wikidata, base de datos de grafos.

El código se encuentra en el repositorio de Github https://github.com/Fran251184/PLN, en la carpeta PLN_TP2_Francisco_J_Alomar, al igual que los archivos *.pdf y el archivo .csv. El algoritmo está implementado en una notebook de Google Colab ya ejecutado, es decir, con los output. Si usted desea ejecutarlo nuevamente, deberá tener una cuenta de Hugging Face y un *toke*n generado. Además, el token tiene que estar guardado como secreto de Colab, bajo el símbolo de llave. Cuando usted ejecute el programa, se solicitará que ingrese el nombre bajo el cual guardó el *token* en los secretos de Colab. De otro modo, no podrá ejecutar el código. La implementación del código y procesamientos de archivos están diseñados para que usted sólo tenga que ingresar su nombre de clave de *token* y el *bot* se activará al final de los procesamientos necesarios, y luego podrá interactuar con el *bot*.

El código está dividido en cuatro secciones: la primera se refiere a la instalación e importación de librerías necesarias para el correcto funcionamiento del programa, la segunda a la solicitud de *token* de Hugging Face, la terca al procesamiento de *.pdf para la construcción de base de datos vectorial, y la cuarta, es la implementación del *bot* experto en deuda externa argentina. De las cuatro secciones profundizaremos en la tercera y cuarta, que son las más importantes. El modo de exposición optado es una descripción general del código y exposición de decisiones tomamos en el diseño. Todas las funciones se encuentran documentadas en el código para aportar mayor claridad a su interpretación y, además, para evitar redundancia en la exposición del informe.

Pues, el desarrollo tiene tres puntos. El primero, en el cual explicamos el procesamiento de texto para base de datos vectorial. Un segundo, donde damos una

explicación general y detallada de la implementación del *bot*. Y un tercero, en el cual indagamos sobre el concepto de *Rerank.* Hacia el final, reflexionamos en las conclusiones sobre el proceso de trabajo y posibles modificaciones que podrían optimizar la *performance* del *chatbot*.

1. Procesamiento de texto para base de datos vectorial

Los archivos *.pdf se descargan directamente del repositorio. Además se crea una carpeta en el entorno de Colab donde se guardan una vez procesados. Esta carpeta se llama archivos_txt_deuda_externa. Para la descarga de los archivos se utiliza la función obtener_enlace_bruto(), la cual arma el enlace bruto de Github. El procesamiento de los archivos *.pdf sigue esta lógica:

- Para cada .pdf se obtiene el enlace bruto.
- De cada .pdf se obtiene el texto mediante la función extraer_texto_sin_tablas_y_gráficos().
- Hacemos una limpieza de subíndices y superíndices del texto.
- Guardamos el texto procesado de los archivos en en la carpeta destino archivos_txt_deuda_externa

La biblioteca que utilizamos para procesar los *.pdf es pdfplumber. Además de convertir a txt y procesar el texto de los pdf, mediante las bibliotecas request y BeautifulSoup hacemos *web scraping* de un artículo de Wikipedia sobre la deuda externa argentina que también es guardado junto a los otros, como .txt en la carpeta destino.

De este modo el algoritmo genera un carpeta con archivos .txt que luego se utiliza para la construcción de base de datos vectorial.

2. Chatbot experto en deuda externa argentina.

Lo primero que se aprecia es un clasificador de *embeddings* del cual no hacemos uso para la implementación del *bot*. Este clasificador es construido mediante la biblioteca sklearn a partir de una serie de ejemplos de *strings* (posibles *querys* de un usuario al modelo), y de ellos se hacen *embeddings* con la biblioteca sentence-transformers, generando así un modelo de regresión logística multinomial que puede clasificar una consulta de usuario en tres clases: EMB, CSV, GRAF. La idea es que mediante un clasificador tal se discrimine una consulta de usuario asociándola a una clase. Esto haría que el modelo LLM utilice de acuerdo a la consulta la base de datos vectorial, tabular o de grafos, respectivamente. Pese a que el modelo (clasificador) parece funcionar bien con muy pocos ejemplos (como puede apreciarse en las métricas y predicciones) hemos tomado la

decisión de descartar el clasificador y hacer uso de LLM para clasificar, por dos motivos principales. En primer lugar, por el trabajo que implica. Para hacer el clasificador multinomial robusto debiéramos aportar más ejemplos de consultas de usuario, mientras que al utilizar zephyr-7b-beta, implementando las técnicas *zero shots* y *few shots*, con pocos ejemplos, se obtiene una *performance* aceptable. En segundo lugar, el modelo LLM nos permite manejar una cuarta categoría de clasificación que, en este caso, sería aquella que no es EMB, CSV ni GRAF. Hemos llamado a esta categoría OTRO. Podríamos incluir a la categoría en el modelo de regresión multinomial, pero debiéramos aportar ejemplos específicos, conjunto difícil de construir dado que contiene todo lo que no son las otras categorías. Por el contrario, como se apreciará más adelante, es mucho más fácil instruir al LLM a clasificar esta posibilidad.

Ahora pasamos a explicar el código específico del *bot*:

- Se carga el modelo de embedding para de HuggingFaceEmbeddings(model_name=nombre_modelo).
- Se descarga la base de datos tabular (.csv) de Github y se genera un *dataframe* (df) con la biblioteca pandas para consultarlo.
- Se cargan los documentos txt desde la carpeta destino.
- Se utiliza Langcahin para dividir el texto. Para ello, se eligieron los siguientes parámetros chunk_size=1000, chunk_overlap=200, separators=["\n\n", "\n", " "].
- A cada documento de la carpeta destino se lo divide según los parámetros anteriores. Se guardan los fragmentos y sus metadatos.
- Se genera la base de datos vectorial, indexada, en Chroma DB con la función configurar_y_cargar_chroma().
- Se generan *embeddings* de strings que nombran a las instituciones internacionales financieras (FMI, BID, Banco Mundial y Club de París)

De este modo, ya se han generado las bases de datos necesarias para que luego las utilice el modelo LLM al construir las respuestas para el usuario. El algoritmo sigue:

- Se imprime un mensaje de presentación del *bot*.
- Se inicia un ciclo *while* que se controla con la entrada por consola de la letra minúscula ‘q’. El usuario debe ingresar ‘q’ para finalizar la conversación.
- Mientras no se ingrese ‘q’ el usuario podrá ingresar un consulta sobre la deuda externa de Argentina.

- Con cada consulta se llama a la función procesar_consulta(), a la cual se le pasan las variables: consulta_usuario, df, embed_model, coleccion, necesarias para procesar las consultas en cada casa.

Podemos decir que la función procesar_consulta() es el “corazón” del bot, dado que desde allí se invoca al clasificador, se clasifica mediante condicionales, y también se llama a cada una de las funciones que son necesarias generar el *prompt* y las respuesta al usuario. Indaguemos un poco esta función:

- Lo primero que se hace es llamar al clasificador mediante clasificar_consulta(), función a la cual se le pasa la consulta del usuario. Esta función consta de un *prompt* particular para clasificación, diseñado con *zero shots* y *few shots*, el cual se pasa generar_clasificacion(). Esta última función genera la respuesta del LLM con parámetros particulares para que su función sea la de clasificar. Esta función devuelve CSV, EMB, GRAF u OTRO. La categoría OTRO, es importante porque como dijimos anteriormente permite manejar la posibilidad de que una query no sea clasificada en ninguna de las tres anteriores, y continuar la conversación con el usuario.
- Una vez hecha la clasificación, se obtiene un embedding de la query del usuario para.

La variable clasificacion, entonces, puede adoptar esos valores. Antes de explicar cada sección de los condicionales, mencionamos algo común a todos ellas. Hemos optado por imprimir la generación de *prompts*, respuesta y todo el procesamiento para hacer *debugging*. Fue de gran utilidad hacer estas impresiones a las cual el usuario no tendría acceso para ir ajustando la interacción con el LLM. Pasamos a describir el primer *if*, cuando la variable clasificacion es igual a CSV:

- Se declara una variable coincidencia_año que extrae de la *query* un número. Si esta variable es vacía, se pasa al *else* que imprime un mensaje para el usuario cómo ingresar el año para saber el monto de deuda y demás información. Si esta variable contiene al menos un número se llama a la función buscar_deuda_anual(), a la que se le pasa el número y el df con los datos tabulares. Si el número coincide con un año del registro del df, se envía esa información al modelo LLM para que construya una respuesta. Si tiene un número no está en el registro se imprime el mismo mensaje que si coincidencia_año está vacía.

Los *prompts* que se envían al modelo LLM, son el mismo para la base de datos tabular que para la base de datos vectorial. Hemos hecho uno específico para cuando la base de datos consultada es de grafos. La generación de respuesta es la misma para las tres bases de datos. Ahora pasamos a describir cuando la variable calcificacion es igual a EMB:

- Se declara resultados = coleccion.query(query_embeddings=[embedding_consulta], n_results=4).
- La función anterior busca en Chroma DB los 4 *embedding* más cercanos por similaridad de coseno a embedding_consulta, que es el *embedding* de consulta del usuario.
- Con el texto que corresponde a estos 4 vectores, se arma el *prompt* sumado a la consulta del usuario como contexto, y el modelo LLM genera una respuesta con esa información.
- A la respuesta que genera el modelo, en este caso, le hacemos una "limpieza". Debido a que en algunos casos el modelo contesta con frases inconclusas, con la función limpiar_respuesta() se sustrae la última frase de la respuesta del modelo si no concluye con punto.

Decidimos generar un prompt particular para la consulta a la base de datos en grafos porque aporta poco contexto al modelo, lo que hacía que conteste con información propia que no provenía de la base de datos. Es por ello que el *prompt* define que se haga una respuesta corta con la información disponible aportada en el contexto. Esto se debe a que el modelo cuenta con información sobre las instituciones internacionales de crédito como el FMI o el BID. Distinto es cuando se pregunta, como en los casos anteriores, por el monto de la deuda para un año o una pregunta histórica específica de la Argentina en relación a la deuda, información con la cual el modelo probablemente no contará. Ahora pasemos a describir cuando clasificacion es igual a GRAF:

- Se declara un diccionario similitudes que asocia códigos de entidades de Wikidata a la similaridad de coseno entre el *embedding* de la *query* del usuario y los *embedding* que se calcularon en un principio para las organización internacionales de crédito.
- De todo ello nos quedamos con el de mayor similaridad, y obtenemos el código de Wikidata.
- Con el código de Wikidata y la biblioteca instalada SPARQLWrapper, hacemos una consulta SPARQL donde de una entidad dada, por ejemplo FMI, consulta a la base

de datos la descripción: “Organismo de las Naciones Unidas Página” y su página oficial “https://www.imf.org/”.

- De este modo se genera un *prompt* y respuesta consecuente por parte del modelo. Es decir, si el usuario pregunta “¿qué es el FMI?”, el modelo contesta con una definición obtenida de la base de datos y proporcionará el link de sitio web oficial para mayor información al respecto.

La función procesar_consulta(), tiene un *else* final que contempla el caso para el cual el modelo no ha podido clasificar la query. Sería cuando la variable clasificacion es igual “OTRO”. En tal caso se imprime un mensaje: "Lo siento, no entiendo tu pregunta. Por favor, proporciona más detalles o intenta con otra pregunta." Finalmente se retorna al ciclo *while* para continuar haciendo consultas o finalizar la interacción con el bot.

3. Concepto de *Rerank*

En el contexto del RAG el *Rerank* es la reclasificación o post-clasificación de documentos que se consideran relevantes, una vez obtenidos ciertos conjuntos de ellos. Justamente, la nueva clasificación es un *ranking* de importancia de los documentos de acuerdo a una consulta dada. Por lo tanto, el *Rerank* es un paso adicional que genera un conjunto más preciso de documentos para generar una respuesta, delimitando otro, previo, que serían los documentos potenciales o susceptibles de ser utilizados. Esta técnica mejora la precisión de la respuesta, reduce el ruido en ella, generando así una optimización del contexto para responder a una *query*. Por lo que pudimos averiguar, se pueden aplicar a las diversas bases de datos y documentos como las que hemos utilizado en el desarrollo del *bot*.

CONCLUSIÓN

Hemos dado una explicación general, aunque también detallada del programa implementado. Debemos resaltar tanto la extensión del código como así también la cantidad de tiempo dedicado. La mayor dificultad fue al principio, a la hora de diseñar el problema a tratar, por un motivo doble. Por una parte, al no conocer en profundidad las herramientas y técnicas que íbamos a utilizar, y, además, por nunca haberlas visto operando de manera integrada, no podíamos elaborar temas pertinentes: ¿cómo diseñar un problema cuando se desconoce el alcance, limitaciones y dificultades de la herramientas para resolverlo? Por otra parte, tal vez por ganar tiempo y querer adelantar resultados, una vez elegido el tema y habiendo implementado código, tuvimos que replantearnos el problema dado que no cumplían con las consignas del tp, es decir, empezamos a trabajar antes que las consignas esten planteadas. Sin embargo, todo el esfuerzo devino en un refuerzo de aprendizaje a

partir del cual advertimos la potencia de los LLMs y la visión integral de herramientas que se requiere para implementarlos con RAG. Además, en perspectiva, este aprendizaje nos permite ahora hacer críticas al programa generado y plantear mejoras.

Lo primero que se advierte es la falta de flexibilidad del código. Hay respuestas que son siempre iguales y poco “humanizadas”, lo cual, entendemos, no sería deseable en un *bot* experto. Para ello, se nos ocurren algunas soluciones. En primer lugar, podríamos plantear un *prompt* y generación de respuesta particular para cuando el clasificador no “comprenda” la pregunta, pero además, que también sea de seguimiento de conversación. Es decir, si el usuario dice “hola”, que el bot no diga “lo siento, no entendí la pregunta...”, sino que responda con saludo e indique cuál es su función. Tal vez, esta instancia podría servir incluso para finalizar la conversación. Otra cuestión de utilidad en este sentido es mantener un historial de la conversación que no implementamos, pero sabemos que se puede hacer, para que haya más coherencia en la conversación y cada pregunta y respuesta no sea independiente de las anteriores. En segundo lugar, podríamos planear la combinación de técnicas de PLN para la búsqueda de datos y construcción de información a partir de las bases de datos. Por ejemplo, no quedarnos solamente con la similaridad de coseno para la búsqueda en bases de datos vectoriales, sino, además, una posibilidad, sería combinarla con *Named Entity Recognition*, herramienta que no utilizamos.

Entendemos que la técnica de *Rerank* sería de gran utilidad, en nuestro caso, justamente, aplicada a la base de datos vectoriales. Por las respuestas que arroja el modelo, advertimos que el modelo toma datos de los textos que no serían relevantes para el contexto, o incluso, son erróneos. Esto se debe a que utilizamos solamente similaridad de coseno para hacer una búsqueda semántica, y si bien hay correlación entre similaridad de coseno y la similaridad semántica, esto no implica que las frases con alta similaridad del coseno sean las más relevantes o precisas para la consulta específica. Dado que la similaridad mide la cercanía de dos vectores, si estos son de alta dimensionalidad, se pueden perder las sutilezas que diferencian a los textos que representan, matemáticamente despreciables, semánticamente relevantes. Por ello, consideramos que aplicar *Rerank* a la elección de embeddings para la consulta de base de datos vectorial podría mejorar la performance de las respuestas del modelo.

Por último, luego de implementar el código, consideramos que la categoría de clasificación podría simplificarse para otorgar mejor precisión. Aunque el modelo resulta robusto con las etiquetas otorgadas, nos parece más simple que las categorías sean más cortas o números, por ejemplo, en vez de CSV, EMB, GRAF y OTRO, que sean 1, 2, 4 y 4, ajustando los parámetros del clasificador consecuentemente.