# Agente Experto en Documento de Seguros

PRUEBA TÉCNICA

Brayan Alejandro Romero Castro

## Introducción.

A continuación se presenta la solución a la prueba técnica proporcionada por Seguros Bolívar para el cargo como profesional de inteligencia artificial. El proyecto-prueba se basa en desarrollar una inteligencia artificial, que sea experto en el siguiente Documento, (el cual es un papper acerca del impacto de la inteligencia artificial en las empresas de seguros). Para ello se diseño este documento explicando la solución, el cual se divide en dos secciones; en la primera se explica el código describiendo cada una de sus partes. Además, se da una manera de presentarlo por medio de Streamlit. En la segunda sección se presenta una manera de subirlo a la nube (en este caso en GCP) para que diferentes personas puedan interactuar con el agente. .

## 1. Desarrollo del agente virtual experto en un entorno local

Para la implementación del agente virtual se decidió utilizar Python, más específicamente Visual Studio Code. Este código implementa una aplicación en Streamlit que permite analizar documentos PDF utilizando LangChain y Ollama. El sistema extrae el texto de páginas específicas de un archivo PDF, procesa una consulta ingresada por el usuario y genera una respuesta utilizando un modelo de lenguaje. Primero, el usuario sube un archivo PDF y selecciona las páginas a analizar; luego, el texto es extraído con la librería PyPDF2. Posteriormente, el usuario ingresa una pregunta sobre el contenido, y LangChain junto con el modelo de lenguaje (llama3.2) generan una respuesta basada en el texto extraído. Además, si se proporciona una respuesta esperada, el sistema calcula la similitud con la generada. El código también incluye registro de eventos (logging) para rastrear errores y medir tiempos de ejecución. Si quiere ver el código completo puede ingresar a este repositorio. Expliquemos cada una de las partes y que hace cada una, empezemos por ver las librerias:

```
import streamlit as st
import PyPDF2
import difflib
import logging
import time
from langchain_community.llms import Ollama
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
```

La función de cada una de estas librerias es la siguiente: streamlit es un framework que se utiliza para crear aplicaciones web interactivas en Python, PyPDF2 se utiliza para leer archivos PDF y extraer su contenido, difflib se usa para calcular la similitud entre textos, logging registra eventos, errores y advertencias en un archivo de logs, time se usa para medir la latencia, Ollama

es un conjunto de modelos de lenguaje (LLM) utilizados para generar respuestas, este se descarga atravez del siguiente link, y en la terminal se ejucta `Ollama run model`, donde model es el modelo que se quiera utilizar en este caso se utilo `llamma3.2`, PromptTemplate y LLMChain, se usan en LangChain para crear prompts y encadenar consultas con el modelo. Ahora veamos a ver las funciones que se implementario en el codigo, comencemo por la función que lee el archivo pdf:

```
def extraer_texto_pdf_paginas(pdf_file, paginas):
    texto = ""
    lector = PyPDF2.PdfReader(pdf_file)
    total_paginas = len(lector.pages)
    for num in paginas:
        if 0 <= num < total_paginas:
            pagina = lector.pages[num]
            texto += pagina.extract_text() + "\n"
        else:
            st.error(f"La página {num+1} no existe en el documento.")
            logging.warning(f"Intento de acceso a página inexistente: {num+1}")
    return texto
```

Esta función toma un pdf y una lista de números de páginas, para convertirlas en un string que luego será entrada para nuestro LLM. Se implementan algunas precauciones como que no se puede introducir un número de páginas que no este en el rango de nuestro pdf y se imprime un anuncio en caso de que llegue a pasar, ahora

```
def main():
    st.title("Análisis de PDF con Langchain, Ollama y Streamlit")
    pdf_file = st.file_uploader("Carga tu archivo PDF", type="pdf")

    if pdf_file is not None:
        lector = PyPDF2.PdfReader(pdf_file)
        total_paginas = len(lector.pages)
        st.write(f"El PDF tiene **{total_paginas}** páginas.")

        paginas_input = st.text_input(
            "Ingresa los números de las páginas que deseas analizar (máx 4):",
            "1,2,3,4"
        )
        try:
            paginas = [int(x.strip()) - 1 for x in paginas_input.split(",") if x.strip().isdigit()]
            if not paginas:
                st.error("No se ingresaron números de página válidos.")
                return
            paginas = paginas[:4]
        except Exception as e:
            st.error(f"Error al procesar las páginas: {e}")
            logging.error(f"Error al procesar las páginas: {e}")
            return

        pdf_file.seek(0)
        texto_seleccionado = extraer_texto_pdf_paginas(pdf_file, paginas)
        if not texto_seleccionado:
```

```
            st.error("No se pudo extraer texto de las páginas seleccionadas.")
            return
        st.success("Se han extraído las páginas seleccionadas correctamente.")
```

La función main() comienza estableciendo la interfaz en Streamlit, donde el usuario puede subir un archivo PDF. Una vez cargado, se utiliza PyPDF2 para leer el número total de páginas del documento, y se le solicita al usuario que ingrese las páginas que desea analizar, se convierte el string que dio el usuarion en una lista, y se verifica que los valores ingresados sean números válidos y se limita la selección a un máximo de cuatro páginas. Luego, el código extrae el texto de las páginas seleccionadas mediante la función `extraer_texto_pdf_paginas`, que habíamos definido anteriormente, asignándole como imput el pdf que el usuario subió y las páginas que selecciono. Durante todo el código se verifica que se accedan imputs validos, como que el número de páginas no sea mayor a 4 (lo cual se hace dado las limitaciones del modelo, puesto que los prompts, que recibe el modelo llama3.2, no van mas allá de 8000 caracteres, que es en promedio lo que este texto tiene por cada 4 páginas), o que el pdf no este vació.

```
        pregunta = st.text_area("Ingresa tu pregunta o instrucción sobre el documento", "")
        ground_truth = st.text_area("Ingresa la respuesta esperada", "")

        if st.button("Procesar consulta"):
            if not pregunta:
                st.error("Por favor, ingresa una pregunta o instrucción.")
            else:
                st.info("Procesando la consulta...")
                logging.info("Procesando consulta con el modelo.")


                llm = Ollama(model="llama3.2")

                prompt_template = PromptTemplate(
                    input_variables=["contexto", "pregunta"],
                    template="""
                    Usa el siguiente contexto para responder la pregunta:
                    Contexto:
                    {contexto}

                    Pregunta:
                    {pregunta}
                    """
                )

                chain = LLMChain(llm=llm, prompt=prompt_template)
```

En esta parte se le pide al usuario que ingrese la pregunta y la respuesta esperada (opcional), por medio de streamlit. Luego de que el procesamiento de la consulta sea valido, cuando el usuario le de al botón procesar consulta, se llamara a Ollama de Langchain y se cargara el modelo llama3.2 cuyas especificaciones puede ver aquí, inmediatamente después se construye la estructura del prompt, mediante PromptTemplate de la cadena (LLMChain), se genera automáticamente un mensaje que

combina el contexto extraído y la pregunta del usuario en un formato predefinido, asegurando que el modelo reciba toda la información necesaria de forma clara y estructurada para generar su respuesta, posterior a esto generamos la instancia chain por medio de la clase LLMchain, tanto el llm (llama3.2) como la estructura del prompt, esto luego sera utilizado al ponerle nombres a las variables contexto y pregunta (cambiandolas por el string de texto sacado del pdf, junto el string representativo de la pregunta que respondio el usario), cabe aclarar que es importante que sea en este orden contexto y luego pregunta dado que la forma en la que actuan los LLM's son generado el ultimo token dependiendo principalmente de lo que tengan inmediatamente atrás, luego si se colocara, primero la pregunta y luego el contexto, no tendria tanta relevancia la pregunta y puede que no se conteste, mas aun que ni siqiera tenga sentido su respuesta. Por ultimo se ejecuta y se evalua todo lo anterior mediante las siguientes lineas de codigo.

```
                # Tiempo de ejecución
                start_time = time.time()
                respuesta = chain.run({"contexto": texto_seleccionado, "pregunta": pregunta})
                end_time = time.time()
                latencia = end_time - start_time

                st.write("### Respuesta:")
                st.write(respuesta)

                st.write(f"**Latencia del modelo:** {latencia:.2f} segundos")
                logging.info(f"Latencia del modelo: {latencia:.2f} segundos")

                if ground_truth:
                    similitud = difflib.SequenceMatcher(None, respuesta, ground_truth).ratio()
                    st.write(f"**Precisión estimada (similitud): {similitud*100:.2f}%**")
                    logging.info(f"Precisión estimada: {similitud*100:.2f}%")

if __name__ == "__main__":
    main()
```

De esta parte destacamos el objeto.método chain.run, de la clase LLMchain, que nos ejecuta la instancia que antes habíamos creado con chain, este tiene como entradas el contexto (el string que generamos del pdf) y la pregunta (el string que introdujo el usuario), esta linea sería la mas importante y la que condensa el objetivo de nuestor llm. Al tiempo que se demore ejecutando esto (el cual medimos con time.time()), será lo que llamaremos latencia de nuestro modelo y que imprimiremos en la pantalla (en promedio se demora unos 35 segundo en responder, mas adelante mostramos prueba de esto). Aparte implementamos la métrica de similitud difflib.SequenceMatcher().ratio(), que nos dice que tan preciso es nuestro modelo, en cuanto a la respuesta que dio con la respuesta que introdujimos debería ser, este ratio, que es un valor entre 0 y 1, se calcula mediante la formula:

$$Ratio = \frac{2 \cdot Numero\_coincidencias}{Longitud\_total} \tag{1}$$

Donde el numero de coincidencias son el número de caracteres en la misma posición o reordenados dentro de una secuencia similar, y la longitud total es la longitud de ambas secuencias

combinadas. Note que es importante que el denominador sea la suma de las longitudes, puesto que si no fuera de esta manera, puede que una respuesta solo por ser mas larga sea mas precisa, lo cual no es necesariamente cierto.

Es importante aclarar las limitaciones de este implementación, lo primero es que al ser un modelo pequeño, no tiene la suficiente capacidad para albergar prompts muy largos, por lo que no va a ser posible leer todo el pdf (esto se podría mejorar comprando Api-keys en Open-AI o implentandolo en una computadora con mas capacidad). Una solución a este problema sin salirse de este esquema es hacer Chunking al pdf, leer chunck (bloque de texto) por chunk con solapamineto (para que no se pierda el contexto), y correr el modelo para cada uno de estos, y que luego la persona seleccione cual es la respuesta que tiene mas sentido, el problema de esta implentación es que es bastante demorada por la cantidad de veces que toca correr el modelo (puede ver una implentación hecha aquí), además por la naturaleza del llm escogido (su poca presición, y su ponderación de importancia), cuando se le hacen preguntas que requieren de contexto que se encuentra muy al principio del pdf seleccionado en la paginas escogidas, puede que se responda de manera difusa y poco precisa. Es por ello que el modelo que se presenta tiene un funcionamiento parcial, es decir, es importante que la persona que lo vaya a utilizar tenga una noción de las paginas en las que podría aparecer la información.

Ahora que ya se sabe como esta implementado el agente, se mostrara algunas pruebas de su funcionamiento, para lo cual se ha tomado el siguiente pdf, el cual a grandes rasgos trata del impacto de la inteligencia artificial en los seguros en un futuro no muy lejano. Lo que haremos es hacer preguntas muy concretas para las cuales podamos medir su nivel de precisión muy bien según la métrica que escogimos en (1), además tomaremos los tiempos de respuesta como se pide en la prueba técnica. Para iniciar el programa escribimos en la terminal de la carpeta en donde se encuentre gardado el codigo: `streamlit run rag_insurance_llm.py`, donde `rag_insurance_llm.py` es el nombre del archivo que contiene el código, luego se nos abrirá la siguiente ventana en donde subiremos el archivo y las páginas que analizaremos, la pregunta y lo que esperamos que nos responda (opcional).

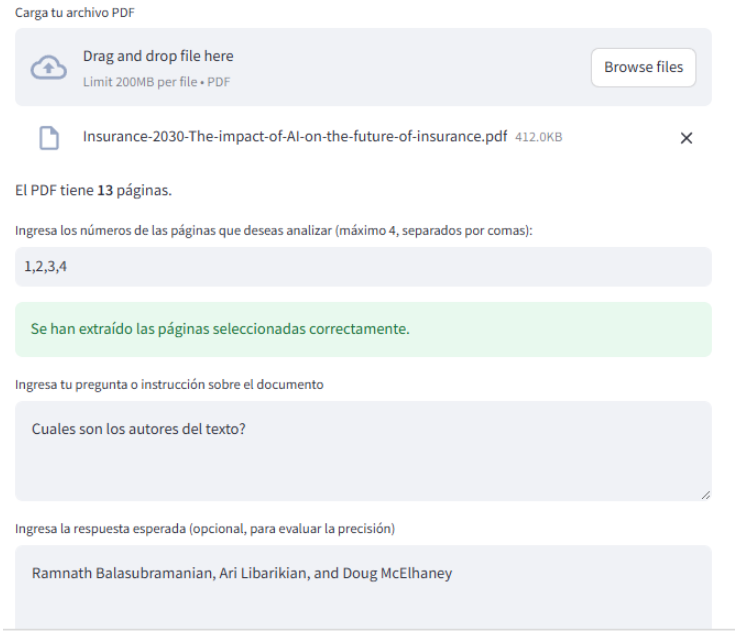


Figure 1: Captura de pantalla del sistema en ejecución.

La respuesta que se obtiene, luego de presionar el botón procesar consulta, es primero el string generado por el LLM (llama 3.2), la latencia o demora del agente en ejecutar la función descrita anteriormente, y la presición o similitud del modelo según la métrica definida en (1), en la imagen vemos que tiene una muy buena presición a pesar de que la respuesta se encuentra muy al principio del Prompt, esto puede pasar al no tener que utilizar ni generar mas tokens de los que ya da el prompt, pero puede pasar que con algunas preguntas un poco mas abstractas y con mas deducción el modelo no de la misma eficiencia, pero si se acerque mucho aunque sea con sinonimos, o las palabras en otro orden.

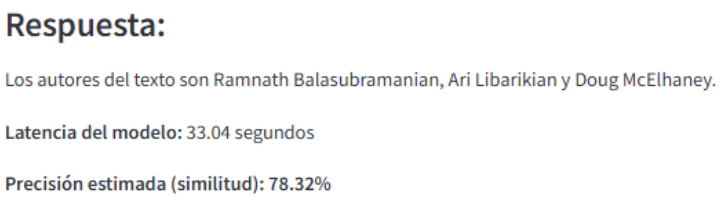


Figure 2: Respuesta del agente

En el siguiente link puede ver mas preguntas y respuestas que se le hizo al agente. A continuación mostramos una gráfica de la presición y tiempo que le tomo para cada uno de las preguntas enumeradas en el link

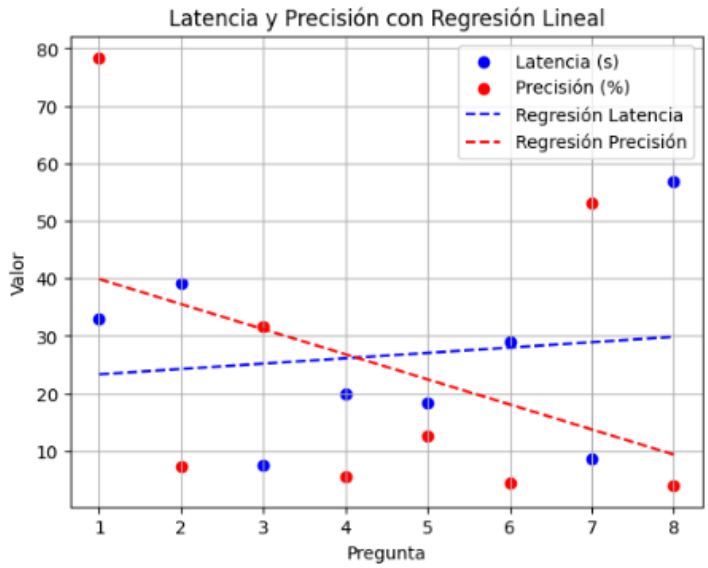


Figure 3: Diagrama de la latencia y tiempo del agente implementado, para diferentes preguntas

A priori los resultados parecen ser desalentadores dada la poca presición del agente para muchas de sus preguntas, pero esto es por la métrica, puesto que las respuestas en su mayor parte son coherentes (como se ve en el siguiete link), solo que en un orden distinto, con sinónimos o con mas texto del que proporciona el usuario (esto hace que el denominador en (1) sea mas grande y por tanto la presición baje, ademas de disminuir la coincidencias al utilizar sinonimos), por ejemplo en la pregunta 2 se le pregunta al agente: *"¿Quien es scott?"*, la respuesta que debería dar sin alargues sería *"Un cliente del año 2030"*, pero el modelo contesta: *"Scott es el protagonista de una histora ficticia presentada en el articulo Insurance 2030-The impact of AI on the future of insurace de Mckinsey and company..."*, es por tanto que la métrica que proporcionamos no es tan buena para el modelo que utilizamos, de esta manera sería mejor que la misma persona evalue que tan buena le ha parecido la respuesta, o implemnetar un LLM que compare las respuestas y diga un numero del 0 al 100 de que tanta relación tienen.

## 2. El agente en la Nube

Si queremos escalar mas este proceso, para utilizar diferentes pdf's y que mas personas tengan acceso a la herramienta, con el modelo que utilizamos, o uno mas potente, lo que se puede hacer es lo siguiente: contenizar con Docker el código en python mostrado en un inicio, y subirlo a Google Cloud Run (que es un servicio gestionado por Google Cloud Plataform, que permite ejecutar aplicaciones contenerizadas sin tener que configuarar la infraestructura subyacente), luego de haber hecho esto contenerizamos el modelo Llama3.2 en Google Kubernetes Engine (GCK), dado que es mas eficiente para este tipo de modelos. Para guardar los archivos pdf utilizaremos Google Cloud Storage (GCS), los cuales se podran escoger una vez iniciemos la aplicación de Streamlit en Cloud Run, El flujo de trabajo seria el siguiente:

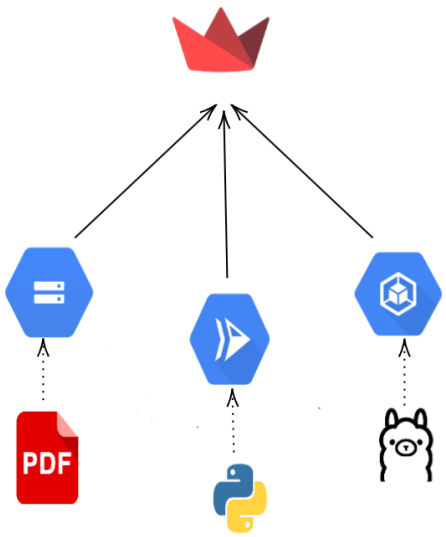


Figure 4: Diagrama de trabajo utilizando las herramientas GCP, experto en documentos

La anterior imagen describe, como estaría estructurado el flujo de trabajo para un agente experto en los pdf's que se le suministren. En la parte superior estaría streamlit, que vendría a ser nuestro frontend y en donde interacutamos con el agente (Este frontend se puede mejorar mas, por ejemplo utilizando Flask, Dash, React). En la parte media tendriamos la nube (sostenida por Google Cloud Plataform, con sus Herramientas GCS, GCR y GCK) en donde se alberga el backend que se desarrollo, en la parte superior de este texto.

**Nota:** La prueba técnica fue desarrollada en su totalidad, y los bonus realizados fueron:

- ✓ Si se quisiera masificar esta solución, con muchos documentos más, qué arquitectura basada en infraestructura en nube sugeriría.
- ✗ Usar GCP para la solución del problema y el punto anterior.
- ✓ Use las librerías para Python como Streamlit para presentar un Front y resolver el punto 2.
- ✓ Usar framework como LangChain o LangGraph para solucionar el punto 1.
- ✓ Presente pruebas de carga de su solución y justifique sus comportamientos.
- ✓ Proponga e implemente una métrica para el seguimiento de la precisión del modelo.
- ✓ Desarrolle logs o seguimiento de la solución capaz de medir los tiempos de latencia de los modelos de LLM.

Por el momento no se alcanzo a configurar los servicios GCS, GCR y GCK, para que el código funcione adecuadamente para mas personas, en cuanto se tenga sera agregado el link a README.md, en este Repositior.

## Bibliografía.

[1] 2018 *Insurance 2030—The impact of AI on the future ofinsurance* Ramnath Balasubramanian, Ari Libarikian, and Doug McElhaney