Bot Bartender Inteligente con Generación Aumentada por Recuperación (RAG) *

Laura Martir Beltrán-C311, Adrián Hernández Castellanos-C312, and Yesenia Valdés Rodríguez-C311

Facultad de Matemática y Computación, Universidad de La Habana (MATCOM)

Este proyecto consiste en el desarrollo de un bot bartender inteligente capaz de responder preguntas sobre cócteles, bebidas y recetas personalizadas. El sistema utiliza una arquitectura basada en RAG (Retrieval-Augmented Generation) para combinar la generación de texto mediante un modelo de lenguaje con la recuperación de información desde una base de datos local y fuentes externas como Difford's Guide.
El objetivo es simular un asistente conversacional experto en coctelería, que no solo responda a preguntas frecuentes, sino que también adapte sus respuestas en función de la disponibilidad de datos, recurriendo a una estrategia de recuperación de conocimiento cuando su conocimiento base no es suficiente.
Este trabajo integra conceptos clave de:
-Inteligencia Artificial (IA): uso de modelos de lenguaje para comprensión y generación de texto.
-Sistemas de Recuperación de Información (SRI): indexación y búsqueda semántica sobre una base de recetas.
-Sistemas Inteligentes y Multiagente (SIM): diseño de un agente autónomo capaz de tomar decisiones sobre cuándo recuperar información, cuándo generar contenido y cómo estructurar sus respuestas.

1. Base de Datos Local

Uno de los requisitos del proyecto establece que "la cantidad total de documentos o ficheros incluidos en el repositorio inicial no debe ser inferior a 2000". Para cumplir con esta condición, se implementó un *crawler-scraper* dirigido al sitio web de Difford's Guide.

Se comenzó analizando el archivo `robots.txt` del sitio, desde el cual se obtuvo acceso a sus *sitemaps*. Con esta información, se recorrieron de forma automatizada todos los enlaces pertenecientes al sitemap cuyo encabezado corresponde al patrón:

```
https://www.diffordsguide.com/cocktails/recipe/
```

De este modo, se garantizó que solo se almacenaran recetas de cócteles, excluyendo otros contenidos del sitio que no son relevantes para el proyecto.

Como resultado del proceso de recolección, se extrajeron un total de 3999 recetas, las cuales fueron almacenadas en una base de datos local en formato `JSON`. Cada entrada presenta la siguiente estructura:

```
{
  "Url": "url de la receta",
  "Name": "nombre del cóctel",
  "Glass": "recipiente en que se sirve",
  "Ingredients": ["lista de ingredientes"],
  "Instructions": "instrucciones de preparación",
  "Review": "detalles extra del cóctel",
  "History": "resumen de la historia del cóctel",
  "Nutrition": "datos nutritivos del cóctel",
  "Alcohol Content": "contenido en alcohol de la bebida",
  "Garnish": "decoración sugerida para el cóctel"
}
```

* https://github.com/LauraMarby/IA-SRI-SIM_Project

El uso del formato `JSON` (*JavaScript Object Notation*) para almacenar las recetas extraídas se fundamenta en su simplicidad, legibilidad y compatibilidad con múltiples tecnologías de procesamiento de datos. JSON es un formato ampliamente adoptado en sistemas de recuperación de información, lo cual facilita su integración posterior con herramientas de análisis, motores de búsqueda, sistemas semánticos y modelos de lenguaje.

Además de estas ventajas operativas, la elección de una estructura bien definida con campos como `Name`, `Glass`, `Ingredients`, `Instructions`, entre otros, responde a una decisión deliberada orientada a la escalabilidad semántica del proyecto. Esta organización jerárquica y semántica de los datos permite una fácil conversión a representaciones más formales como *ontologías*, utilizando estándares como OWL.
Por tanto, no solo se adoptó JSON por su practicidad, sino también como un primer paso hacia la representación del conocimiento de manera estructurada y semánticamente enriquecida.

2. Repositorio Vectorial (Embeddings):

A partir de los documentos extraídos, se construyó un repositorio vectorial con el objetivo de representar el contenido textual en un espacio de alta dimensionalidad que permita realizar búsquedas semánticas eficientes.

Para ello, cada documento fue preprocesado utilizando la técnica de *ventanas deslizantes* (*sliding window*). Este método consiste en dividir el texto en fragmentos (*chunks*) de tamaño fijo, definidos por el parámetro `window_size`, cuyo valor por defecto es 100 tokens. Estos fragmentos se generan con un solapamiento determinado por el parámetro `stride`, establecido por defecto en 60 tokens. Esta estrategia permite conservar la coherencia contextual entre los fragmentos generados, minimizando la pérdida de información semántica entre ventanas consecutivas.

Cada uno de estos fragmentos fue posteriormente transformado en un vector mediante un modelo de embeddings preentrenado, específicamente `all-MiniLM-L6-v2`. Este modelo fue utilizado a través de la clase `SentenceTransformer` de la biblioteca `sentence_transformers`, la cual se obtuvo desde la plataforma Hugging Face.
El resultado de este proceso es un repositorio vectorial que sirve como base para la etapa de recuperación semántica en el sistema de RAG (Retrieval-Augmented Generation), permitiendo emparejar consultas del usuario con los fragmentos de texto más relevantes de forma eficiente y significativa.

El repositorio resultante, compuesto por los vectores y sus fragmentos textuales asociados, fue almacenado en un archivo con extensión `.pkl` (*Pickle*). Esta decisión se tomó por las siguientes razones: el formato `Pickle` permite una serialización eficiente de objetos complejos de Python, conserva la estructura original de los datos (incluyendo listas, diccionarios, matrices NumPy, entre otros), y permite una carga rápida en memoria durante la ejecución del sistema. Gracias a esta elección, se optimizó tanto el tiempo de inicialización del sistema como la reutilización de los datos procesados sin necesidad de recálculo, lo cual es especialmente importante en flujos de trabajo iterativos como los que se presentan en aplicaciones basadas en RAG.

3. Ontología:

Para modelar y representar de forma semántica el conocimiento extraído de las recetas de cócteles, se desarrolló una ontología utilizando la biblioteca `OWLReady2` en Python. Esta biblioteca fue seleccionada por su integración nativa con objetos de Python, su soporte completo para OWL 2, y la facilidad que ofrece para definir clases, propiedades y axiomas, así como para guardar y cargar ontologías en formatos estándar como RDF/XML.

Clases principales:

Las clases centrales de la ontología definen las entidades fundamentales del dominio de los cócteles:

- **Cocktail**: Representa las distintas recetas de cócteles.
- **Ingredient**: Modela los ingredientes utilizados en las recetas.
- **Glass**: Describe los tipos de vasos en los que se sirven los cócteles.
- **AlcoholContent**: Contiene información relativa al contenido de alcohol de la bebida.

Propiedades de objeto:

Las propiedades de objeto permiten establecer relaciones entre distintas clases de la ontología:

- **hasIngredient**: Relaciona un cóctel con uno o más ingredientes.
- **servedIn**: Establece el tipo de vaso en el que se sirve un determinado cóctel.
- **hasAlcoholContent**: Vincula un cóctel con su información de contenido alcohólico.

Propiedades de datos:

Las propiedades de datos definen atributos específicos asociados a las instancias de las clases:

- Para la clase **Cocktail**:
  - **hasName**: Nombre del cóctel.
  - **hasUrl**: URL donde se encuentra la receta.
  - **hasInstructions**: Instrucciones para su preparación.
  - **hasReview**: Comentarios o reseñas adicionales.
  - **hasHistory**: Información histórica o de origen del cóctel.
  - **hasNutrition**: Información nutricional de la bebida.
  - **hasGarnish**: Elementos decorativos del cóctel.
- Para la clase **AlcoholContent**:
  - **standardDrinks**: Número de bebidas estándar.
  - **alcoholVolume**: Porcentaje de volumen de alcohol.
  - **pureAlcohol**: Cantidad de alcohol puro.

Exportación y formato:

La ontología fue exportada en el formato `RDF/XML`, ya que este es ampliamente aceptado como estándar de serialización en entornos semánticos y facilita su interoperabilidad con herramientas como Protégé, sistemas de razonamiento lógico, y consultas SPARQL. Este formato garantiza la compatibilidad con otros componentes semánticos que puedan integrarse en futuras ampliaciones del sistema.

4. Modelo de Lenguaje Utilizado:

Para el desarrollo del proyecto y la ejecución de pruebas experimentales se empleó el modelo de lenguaje `gemini-1.5-flash`, provisto por la plataforma Google AI. El acceso a este modelo se realizó mediante una `API key` y fue gestionado a través de la clase `GenerativeModel`, perteneciente al módulo `genai` de la biblioteca `google.generativeai`.

La elección de este modelo responde, en primer lugar, a su capacidad de respuesta rápida y eficiente, optimizada para tareas de generación aumentada por recuperación (RAG). Además, su uso había sido previamente introducido en sesiones prácticas del curso, lo que facilitó su integración y redujo la curva de aprendizaje. Esta familiaridad previa permitió concentrar los esfuerzos del equipo en los aspectos técnicos y metodológicos del proyecto, sin incurrir en tiempos adicionales de adaptación a nuevas herramientas.

5. Espacio de Sabores (Flavor Space):

Con el objetivo de incorporar lógica difusa y lógica n-valente en la interpretación de preferencias subjetivas expresadas por el usuario, se diseñó un *espacio de sabores* basado en variables lingüísticas. Esta estructura semántica permite modelar descripciones cualitativas de preferencias gustativas y convertirlas en valores cuantificables para la toma de decisiones en el sistema.

Las variables lingüísticas consideradas en este espacio fueron: `Dulce`, `Salado`, `Amargo`, `Ácido` y `Picante`. Cada una de estas dimensiones se representa mediante un conjunto de etiquetas difusas: `Nada`, `Poco`, `Medio` y `Muy`, cuyos rangos se definen dentro del intervalo $[0,1]$ de la siguiente manera:

- Función de membresía rectangular izquierda:

$$\mu_{\text{nada}}(x) = \begin{cases} 1 & \text{si } x \leq 0{,}1 \\ 0 & \text{si } x > 0{,}1 \end{cases}$$

- Función de membresía triangular:

$$\mu_{\text{poco}}(x) = \begin{cases} 0 & \text{si } x \leq 0{,}1 \text{ o } x \geq 0{,}4 \\ \frac{x-0{,}1}{0{,}15} & \text{si } 0{,}1 < x < 0{,}25 \\ \frac{0{,}4-x}{0{,}15} & \text{si } 0{,}25 \leq x < 0{,}4 \end{cases}$$

- Función de membresía triangular:

$$\mu_{\text{medio}}(x) = \begin{cases} 0 & \text{si } x \leq 0{,}3 \text{ o } x \geq 0{,}7 \\ \frac{x-0{,}3}{0{,}2} & \text{si } 0{,}3 < x < 0{,}5 \\ \frac{0{,}7-x}{0{,}2} & \text{si } 0{,}5 \leq x < 0{,}7 \end{cases}$$

- Función de membresía trapezoidal derecha:

$$\mu_{\text{muy}}(x) = \begin{cases} 0 & \text{si } x < 0{,}6 \\ \frac{x-0{,}6}{0{,}2} & \text{si } 0{,}6 \leq x < 0{,}8 \\ 1 & \text{si } x \geq 0{,}8 \end{cases}$$

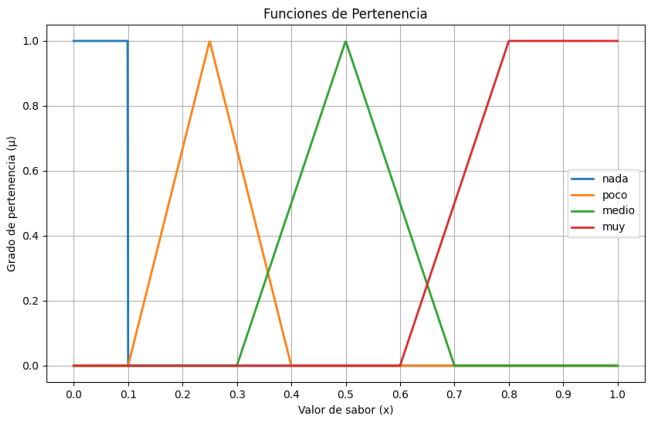

Figura 1. Funciones de pertenencia difusa: nada, poco, medio, muy.

Este modelo difuso permite responder de manera flexible a consultas del tipo: "quiero algo poco dulce y muy ácido", asignando grados de pertenencia a cada sabor y combinando resultados en función de las similitudes encontradas en la base de datos de cócteles. Así, el sistema puede identificar recetas cuyos perfiles de sabor se ajusten parcialmente a las preferencias del usuario, aun cuando no exista una coincidencia exacta.

5.1. Asignación de Sabores a Ingredientes:

Para cada ingrediente presente en las recetas almacenadas en la base de datos local, se realizó una consulta al modelo de lenguaje con el objetivo de estimar su perfil gustativo. En particular, se solicitó al modelo que asignara un vector de pertenencia correspondiente a los cinco sabores definidos en el espacio de sabores: dulce, salado, amargo, ácido y picante.
Cada vector generado representa los grados de membresía del ingrediente a cada una de las categorías gustativas, en el siguiente orden:

1. Posición 0: grado de membresía a `Dulce`,
2. Posición 1: grado de membresía a `Salado`,
3. Posición 2: grado de membresía a `Amargo`,
4. Posición 3: grado de membresía a `Ácido`,
5. Posición 4: grado de membresía a `Picante`.

Sin embargo, se observó que el modelo de lenguaje presenta limitaciones al interpretar ingredientes poco comunes o específicos, como por ejemplo `Ketel One Citroen Vodka`. En estos casos, la asignación automática de los grados de pertenencia no resultaba fiable o coherente. Para garantizar la calidad del sistema, fue necesario intervenir manualmente en la anotación de dichos ingredientes, asignando vectores de sabor basados en conocimiento experto o descripciones obtenidas de fuentes externas.
Este proceso permitió completar el espacio semántico de ingredientes de manera robusta, asegurando la consistencia de los vectores utilizados en la inferencia difusa de sabores.

5.2. Inferencia de Sabores a Nivel de Cóctel:

Una vez obtenidos los vectores de membresía para cada ingrediente presente en la base de datos, se procedió a realizar la inferencia lógica correspondiente a nivel de cóctel, con el fin de estimar el perfil gustativo de cada receta en función de la composición de sus ingredientes.

Para este proceso se implementó un enfoque basado en el modelo difuso de Takagi–Sugeno–Kang (TSK), el cual permite definir reglas cuyo resultado es una operación matemática sobre las variables de entrada. En este contexto, se asumió que el grado de pertenencia de un cóctel a un sabor determinado puede ser modelado como el promedio de los grados de membresía de ese mismo sabor entre todos sus ingredientes.
Formalmente, si un cóctel C está compuesto por n ingredientes $I_1, I_2, \ldots, I_n$, y si $\mu_s(I_k)$ denota el grado de pertenencia del ingrediente I_k al sabor $s \in \{\text{Dulce, Salado, Amargo, Ácido, Picante}\}$, entonces el grado de pertenencia del cóctel al sabor s, denotado como $\mu_s(C)$, se calcula como:

$$\mu_s(C) = \frac{1}{n} \sum_{k=1}^{n} \mu_s(I_k) \tag{1}$$

Este método garantiza una representación agregada coherente del perfil gustativo de cada cóctel, basada en la contribución individual de sus ingredientes y respetando la estructura semántica del espacio de sabores definido previamente.

6. Sistema Multiagente:

El sistema desarrollado está basado en una arquitectura multiagente, en la cual múltiples componentes autónomos interactúan y cooperan entre sí para proporcionar respuestas eficientes y coherentes a las consultas del usuario. Cada agente posee responsabilidades específicas y se comunica con los demás mediante protocolos establecidos a través de un canal previamente definido para este propósito. Esta organización modular favorece la escalabilidad, facilita el mantenimiento y permite una futura ampliación del sistema de manera estructurada.

6.1. Agentes que Componen el Sistema:

A continuación, se describen brevemente los agentes que conforman el sistema y sus respectivas funciones:

- **Agente Usuario:**
  Este agente actúa como la interfaz de entrada del sistema, siendo el encargado de recibir la consulta textual proporcionada por el usuario final. Una vez iniciada su ejecución, espera la entrada del usuario y, en función de su contenido, toma una decisión: finalizar la interacción o continuar con el procesamiento de la solicitud.
  En caso de continuar, el agente Usuario envía la consulta al *Agente Detector de Intenciones*, quien determina la intención subyacente de la misma. Posteriormente, tanto la consulta original como la intención detectada son remitidas al *Agente Coordinador*, que se encarga de orquestar el procesamiento distribuido.
  Una vez que el Coordinador genera una respuesta final, esta es devuelta al Agente Usuario, quien la presenta al usuario a través de la interfaz correspondiente. Este flujo asegura una interacción fluida y estructurada entre el sistema y el usuario final.
- **Agente Detector de Intenciones:**
  El Agente Detector de Intenciones tiene como objetivo identificar de forma precisa el propósito de la consulta del usuario. Para ello, se apoya en un modelo de lenguaje de propósito general que recibe un *prompt* estructurado y devuelve una respuesta en formato JSON que permite categorizar la intención expresada.
  Este agente no solo identifica si se ha mencionado un cóctel específico, sino que también traduce la consulta al inglés, determina si se está solicitando información sobre sabores particulares, y genera preguntas complementarias para ser consultadas en el repositorio vectorial.
  La estructura del *prompt* enviado al modelo de lenguaje está cuidadosamente diseñada para capturar todos los elementos relevantes de una consulta. A continuación se muestra un fragmento del código en Python utilizado para la construcción del mensaje:

```
prompt = f"""
Eres un asistente para preprocesar consultas de usuarios sobre c\'octeles.

Dada esta consulta: ‘‘‘{query}‘‘‘

Responde en JSON con las siguientes claves:

- "original_language": el idioma original del usuario.
- "translated_prompt": la traducci\'on al ingl\'es de la consulta.
- "cocktail_mentioned": true o false.
- "cocktails": una lista con un objeto por cada c\'octel mencionado.
  Cada objeto debe tener:
    - "name": nombre del c\'octel.
    - "fields_requested": lista de 9 valores booleanos correspondientes a:
      Url, Glass, Ingredients, Instructions, Review,
      History, Nutrition, Alcohol_Content, Garnish.
- "flavor_of_drink": tragos que deben tener un perfil de sabor espec\'ifico.
- "embedding_query": preguntas que deben resolverse con el repositorio vectorial.
```

```
- "fields": True o False, seg\'un si la informaci\'on se puede obtener directamente
  de los campos disponibles del c\'octel.
"""
```

Listing 1.1. Construcción del prompt para detección de intenciones

La clave `fields_requested` representa una lista ordenada de valores booleanos, cada uno correspondiente a un campo específico del cóctel, definidos por la constante:

```
CAMPOS_TRAGO = [
    "Url", "Glass", "Ingredients", "Instructions",
    "Review", "History", "Nutrition", "Alcohol_Content", "Garnish"
]
```

Listing 1.2. Campos disponibles para cada cóctel

Este enfoque permite extraer de forma estructurada la intención del usuario, incluso si la consulta está incompleta, poco clara o redactada en un lenguaje natural complejo.

- **Agente Coordinador:** El agente coordinador actúa como el núcleo del sistema multiagente, encargado de gestionar el flujo de información y coordinar la interacción entre los distintos componentes. Su objetivo principal es procesar la consulta del usuario, distribuirla a los agentes especializados según la intención detectada, y ensamblar las respuestas parciales en una respuesta final coherente.
  Sus funciones principales incluyen:
  - **Procesamiento de consultas:** Limpia y analiza la consulta en formato JSON, extrayendo parámetros clave como el idioma original, el `translated_prompt`, el `embedding_query`, los cócteles solicitados y las preferencias de sabor.
  - **Distribución de tareas:** En función de la intención identificada, decide qué agentes deben intervenir: ontología (conocimiento estructurado), embeddings (búsqueda semántica), sabores (preferencias gustativas) o crawler (fuentes externas).
  - **Gestión de mensajes:** Recibe y direcciona mensajes desde el usuario, el validador y el crawler, manteniendo un estado interno con la información relevante de la sesión.
  - **Construcción de respuestas:** Evalúa la información recuperada por los agentes y, con base en la validación, decide si generar una respuesta final, solicitar una búsqueda en línea adicional o devolver una respuesta parcial. También es responsable de generar prompts detallados para el modelo de lenguaje y de presentar la salida de forma amigable.

    ```
    prompt = f"""
        You are an expert bartender assistant.
        Answer the following user query in a clear, helpful, and friendly way
        Query: "{self.query}"
        You previously selected the following answer as most relevant:
        \"\"\"{respuesta}\"\"\"
        Additional helpful data:
        \"\"\"{complementos}\"\"\"
        The reasoning used to choose this answer:
        \"\"\"{razonamiento}\"\"\"
        Now, write a final answer in {self.lang.upper()} to send to the user, using the selected
            data and reasoning. Do not mention that it came from a model or that it was selected.
            Just provide the final, helpful answer.
        If you see in the reasoning something like "Informaci\'on al respecto no encontrada.
            Realizando b\'usqueda online." You should mention to the user that his request is being
             searched online and that he should wait.
        On that case, the user should wait for your answer and will not be able to insert an input,
             so don't try to tell him to do anything else but wait.
        """
    ```

    Listing 1.3. Construcción del prompt para generación de respuesta final.

  - **Manejo de errores:** Implementa mecanismos de control ante casos excepcionales o cuando la información recuperada es insuficiente.

Este agente constituye el "cerebro" del sistema, tomando decisiones estratégicas sobre qué información buscar, cómo integrarla y cuándo responder, garantizando una experiencia conversacional efectiva y coherente para el usuario.

- **Agente de Embedding:**
  Este agente es el encargado de realizar consultas sobre el repositorio vectorial generado previamente a partir del corpus de recetas, utilizando técnicas de *Retrieval-Augmented Generation* (RAG). Su función principal consiste en encontrar recetas de cócteles semánticamente relevantes a partir de la consulta del usuario.

  Para lograrlo, se emplea una búsqueda por similitud basada en la **distancia euclidiana** entre los vectores de embedding de los textos almacenados y el de la consulta. Esta métrica se define formalmente como:

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} \tag{2}$$

  donde $\mathbf{u}$ y $\mathbf{v}$ son los vectores de embedding en un espacio de dimensión n.
  La elección de la distancia euclidiana se justifica por su simplicidad computacional y su efectividad en espacios de representación densa como los generados por modelos de lenguaje preentrenados. En particular, se considera apropiada cuando los vectores están normalizados o cuando se desea preservar relaciones de proximidad geométrica.

  Una vez identificados los k fragmentos de texto más cercanos a la consulta, estos se envían al *Agente Validador* para su posterior análisis y validación semántica, contribuyendo así a la generación de una respuesta coherente y contextualizada para el usuario.
- **Agente Ontológico:**
  Este agente utiliza una ontología semántica centrada en el dominio de los cócteles y sus atributos, con el objetivo de representar conocimiento estructurado de manera formal. Gracias a esta representación, es posible realizar razonamientos complejos, como inferencias sobre relaciones jerárquicas entre ingredientes, utensilios, tipos de cócteles y técnicas de preparación. Su función principal es consultar la ontología para extraer la información relevante según la intención detectada en la consulta del usuario, sirviendo como fuente estructurada y confiable de conocimiento dentro del sistema.
- **Agente de Sabores:**
  Este agente implementa un mecanismo basado en lógica difusa para evaluar la correspondencia entre las preferencias gustativas expresadas por el usuario y los perfiles de sabor predefinidos de cada cóctel. El proceso se apoya en vectores de pertenencia que representan, para cada bebida, su grado de afinidad con distintas categorías de sabor (por ejemplo, dulce, amargo, ácido).
  La consulta del usuario se traduce a una fórmula lógica compuesta por operadores `AND` y `OR`, la cual es transformada automáticamente a su forma normal disyuntiva (DNF) mediante la función `to_dnf` del módulo `sympy` en Python. Esto permite recorrer de manera estructurada las disyunciones (cláusulas) y sus respectivas conjunciones (términos) para evaluar los grados de cumplimiento.

  Evaluación Lógica Difusa Para cada cóctel c y para cada cláusula ϕ de la fórmula en DNF, se calculan los grados de pertenencia $\mu_i(c)$ con respecto a las categorías de sabor i involucradas. La interpretación lógica se realiza bajo el paradigma de la lógica difusa:

  - El operador `AND` se modela con el mínimo (mín) de los grados de pertenencia.
  - El operador `OR` se modela con el máximo (máx) de los resultados de cada cláusula.

Así, el grado total de cumplimiento de un cóctel c con respecto a la fórmula lógica F es:

$$\mu_F(c) = \max_{\phi \in \text{DNF}(F)} \left(\min_{i \in \phi} \mu_i(c) \right)$$

Filtrado por Tolerancia y Selección Probabilística Una vez evaluados todos los cócteles, se seleccionan aquellos cuyo grado de cumplimiento sea mayor o igual a un umbral de tolerancia definido (por ejemplo, $\tau = 0{,}7$). A partir de los valores $\mu_F(c)$ de los cócteles aceptados, se aplica una función **softmax** para convertir estos valores en una distribución de probabilidad:

$$P(c_i) = \frac{\exp\left(\mu_F(c_i)/T\right)}{\sum_j \exp\left(\mu_F(c_j)/T\right)}$$

donde $T > 0$ es un parámetro de temperatura que controla la dispersión de la distribución. En este proyecto tomaremos para T el valor de 0.5 estático para mantener una varianzza equilibrada.
A partir de esta distribución, se genera una muestra aleatoria sin repetición de k cócteles, priorizando aquellos con mayor grado de satisfacción según los criterios de sabor definidos por el usuario.

Ventajas Este enfoque permite un filtrado flexible y adaptativo, integrando razonamiento lógico, incertidumbre y aleatoriedad, lo que enriquece la experiencia personalizada del usuario y simula un proceso de recomendación más humano y contextualizado.

- **Agente Validador:**
  Este agente actúa como un filtro inteligente que garantiza la calidad, coherencia y relevancia de las respuestas generadas por el sistema. Su función es validar que cada respuesta candidata cumpla con los criterios esperados antes de ser entregada al usuario, asegurando consistencia, completitud y adecuación al formato.

  Funciones Principales
  - **Gestión de Mensajes:** Recibe resultados de otros agentes junto con las expectativas (fuentes esperadas). Una vez que se han recolectado todos los datos necesarios, se inicia el proceso de validación.
  - **Extracción de Restricciones:** Utiliza un modelo de lenguaje para identificar restricciones de la consulta del usuario, clasificándolas en:
    - **Fuertes:** Requisitos explícitos (ej. “debe contener vodka”).
    - **Débiles:** Preferencias opcionales (ej. “preferiblemente frutal”).
    - **Conjuntas:** Restricciones globales (ej. “mínimo tres tragos”).

    El resultado se estructura como un objeto JSON.

```
    prompt = f"""
Eres un asistente para procesar consultas de usuarios sobre cocteles y tragos. Dada esta
    consulta de usuario: \"{query\}"

Extrae las restricciones de contenido que debe cumplir una buena respuesta. Estas restricciones
     deben clasificarse en fuertes y debiles.
Las restricciones fuertes son aquellas que garantizan todo lo que se menciona explicitamente en
     la pregunta respecto a un trago o tragos, y debe ser un conjunto peque\~no de
    restricciones separadas, representando cada problematica a resolver de la pregunta. Cada
    restriccion debe ser un texto lo mas basico y peque\~no posible. No incluya restricciones
    de tragos conjuntos, separe en problematicas por cada trago."
Las restricciones debiles son aquellas que no se tienen que cumplir necesariamente, pero aporta
     valor a la calidad de la respuesta. Debe ser un conjunto peque\~no tambien, de
    restricciones separadas. No incluya restricciones de tragos conjuntos, separe en
    problematicas por cada trago."
```

```
Cada restriccion debe estar orientada exclusivamente a un problema que debe cumplirse respecto
    a un unico trago. Ademas, cada candidato de respuesta corresponde tambien a un unico trago
    , asi que ningun documento contendra independientemente varias recetas sobre varios tragos
    , por lo cual este tipo de restricciones debe quedar situada en otro campo.
Devuelva un JSON con el siguiente formato:\n
fuertes: [Todas las restricciones fuertes respecto a cada trago. Esto debe ser una lista de
    string]\n
debiles: [Lista o conjunto de restricciones debiles. Esto debe ser una lista de string]\n
conjuntas: [Conjunto de restricciones que se debe cumplir en general, y depende de la cantidad
    de tragos y especificacion de los mismos que requiere la pregunta.]
NOTA: Ninguna de las restricciones extraidas debe ser algo ambiguo, tienen que ser afirmaciones
     facilmente verificables y explicitas. Ademas, ignora restricciones que no traten
    explicitamente sobre la bebida en cuestion.
"""
```

Listing 1.4. Construcción del pormpt para extracción de restricciones.

- **Verificación de Cumplimiento:** Para cada candidato, se genera una matriz booleana indicando el cumplimiento de las restricciones. Esta evaluación también se realiza mediante prompt especializado a un modelo de lenguaje.

```
    prompt = f"""
        Tenemos una lista de respuestas candidatas y una lista de restricciones.
        Para cada respuesta, indica si cumple cada restriccion (si o no).
        Responde en formato JSON como una lista de objetos, uno por respuesta.
        Cada objeto debe tener esta estructura:
        {{"respuesta": "texto", "cumple": ["si", "no", "si", ...]}}
        Respuestas:
        {json.dumps(respuestas[:200], ensure_ascii=False)}
        Restricciones:
        {json.dumps(restricciones, ensure_ascii=False)}
        """
```

Listing 1.5. Construcción del pormpt para la verificación de la matriz.

- **Selección de Candidatos:** Se aplica la metaheurística `TabuSearchSelector` para elegir las respuestas óptimas, basándose en un puntaje de fitness que pondera el cumplimiento de restricciones fuertes, débiles y globales.
- **Verificación de Suficiencia:** Se evalúa si la información disponible satisface adecuadamente la consulta del usuario. Esta etapa considera tanto datos principales como secundarios, y determina si es necesaria una búsqueda en línea.

```
    prompt = f"""
        Tengo una pregunta que realizo un usuario, y tengo informacion que puedo usar para
            conformar una respuesta.

Pregunta del usuario:
"{pregunta}"

Informacion que considero importante:
{[r for r in respuestas]}

Informacion que considero util y secundaria:
{[r for r in candidates]}

Ademas, estas son las restricciones que debe cumplir una buena respuesta, ademas de las que
    puedas inferir por la pregunta del usuario:
{[r for r in restricciones_grupales]}

Tu respuesta debe ser un OBJETO JSON con la siguiente estructura exacta, usando solo tipos
    validos de JSON (booleanos, listas, cadenas, etc.), **sin poner todo como texto ni
    concatenar campos**:

‘‘‘json
{{
```

```
  "suficiente": Indica si con la informacion importante es suficiente o no dar una respuesta
      correcta que cumpla las restricciones.
  "expandida_suficiente": Indica si agregando algunos de los fragmentos extra, es suficiente o
      no dar una respuesta correcta que cumpla las restricciones.
  "razonamiento": "..."
  "requiere_busqueda_online": True o False. Esto depende del contexto de la pregunta. Por
      ejemplo si me piden un trago que no encuentro, esto debe ser True, pero si me piden algo
      como (Recomiendame algo) sin dar detalles de sus gustos, esto debe ser falso porque
      cualquier trago podria gustar.
  **NOTA IMPORTANTE: Si la pregunta del usuario implica explicitamente una busqueda en internet
      , requiere_busqueda_online debe ser True.**
}}
"""
```

Listing 1.6. Construcción del pormpt para la verificación de suficiencia.

Utilidades Internas El agente cuenta con herramientas de normalización de candidatos, eliminación de duplicados y manejo robusto de errores, incluyendo extracción manual de respuestas desde estructuras JSON malformadas.

Observaciones Finales El diseño actual sustituye la heurística original basada en optimización por colonia de hormigas por una búsqueda tabú, mejorando la eficiencia en la selección de respuestas. Este agente prioriza la calidad sobre la cantidad, asegurando que la interacción del usuario con el sistema sea precisa, relevante y útil.

- **Agente Crawler:**
  Este agente se encarga de la extracción dinámica de información desde fuentes externas cuando el conocimiento local del sistema resulta insuficiente para generar una respuesta adecuada.

  Mecanismo de búsqueda Al recibir una consulta del usuario que requiere información externa, el agente construye una query basada en dicha solicitud y la procesa utilizando la función `search` del módulo `googlesearch`. Se recupera el primer resultado disponible, asumiendo que representa la fuente más relevante, especialmente en consultas dirigidas a un solo cóctel.

  Filtrado de fuentes confiables Para garantizar la calidad y coherencia del conocimiento integrado, el agente únicamente acepta resultados provenientes de sitios verificados y especializados en coctelería:

  - Difford's Guide
  - Liquor
  - Punch Drink

  Este criterio evita la incorporación de contenido irrelevante, como discusiones informales en foros o sitios no especializados que puedan degradar la calidad del repositorio.

  Integración eficiente al sistema Una vez obtenida una receta válida, esta se serializa en formato JSON y se agrega a la base de datos local. Además, se procesa mediante técnicas de *embedding* para generar su representación vectorial semántica. En lugar de recalcular el repositorio completo, los nuevos vectores se concatenan al repositorio vectorial existente. Esta estrategia evita cómputos innecesarios y mejora significativamente el rendimiento del sistema en tiempo de consulta.

6.2. Flujo:

A continuación, una representación de cómo funciona el flujo del sistema.

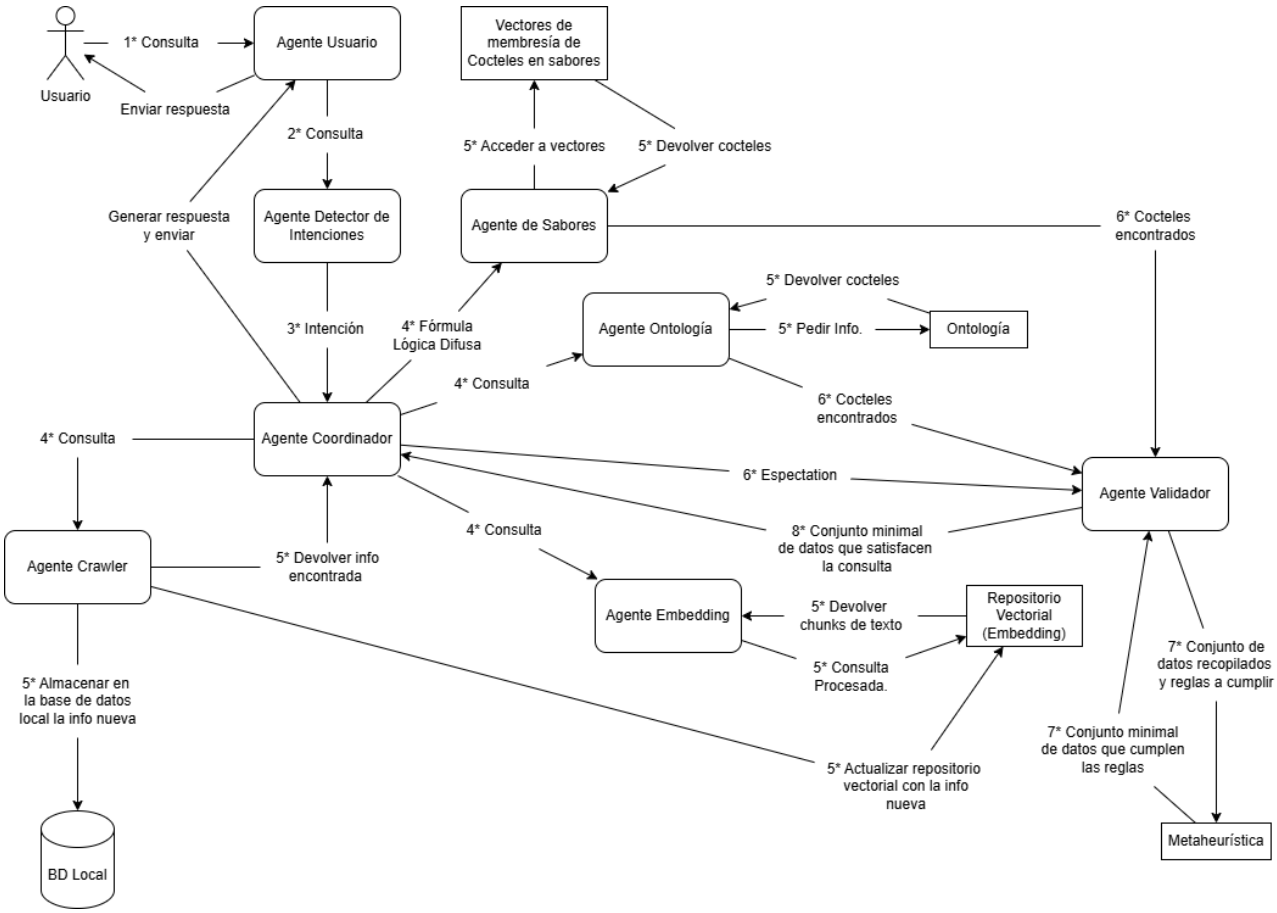

Figura 2. Flujo del sistema.

7. Tests realizados y resultados obtenidos

Con el objetivo de validar la funcionalidad, robustez y rendimiento de los componentes principales del sistema, se llevaron a cabo diversos tests controlados. Estas pruebas permiten no solo verificar la precisión individual de cada agente, sino también evaluar el comportamiento global del sistema bajo condiciones realistas.

7.1. Testeando la Ontología

Se seleccionaron aleatoriamente 1000 cócteles del repositorio local y se realizaron consultas sobre la ontología semántica diseñada. Esta prueba tiene como propósito confirmar la integridad estructural de la ontología, así como su capacidad de recuperar información precisa y completa para cada entidad.

```
Resultados de validación de ontología:
 - Url              : 100.00\% campos correctos
 - Glass            : 100.00\% campos correctos
 - Ingredients      : 100.00\% campos correctos
 - Instructions     : 100.00\% campos correctos
 - Review           : 100.00\% campos correctos
 - History          : 100.00\% campos correctos
 - Nutrition        : 100.00\% campos correctos
 - Alcohol_Content  : 100.00\% campos correctos
 - Garnish          : 100.00\% campos correctos

Total de tragos válidos: 1000 / 1000
Errores de carga/consulta: 0
```

Los resultados muestran una cobertura total, sin errores de carga o recuperación, lo que evidencia la fiabilidad y consistencia del modelo ontológico.

7.2. Testeando el Embedding

Se seleccionaron 1000 documentos aleatorios y, para cada uno, se extrajeron hasta 10 fragmentos representativos. A cada fragmento se le introdujo ruido mediante un modelo de lenguaje, que generó descripciones ligeramente alteradas. Estas descripciones se utilizaron como queries sobre el sistema de recuperación basado en embeddings, validando si el documento original se encontraba entre los más relevantes.

```
Resultados de validación de embedding:
Total evaluaciones: 4550
Precisión en top-1: 29.49\%
Precisión en top-5: 30.81\%
```

A pesar de la complejidad introducida por el ruido semántico, el sistema logra recuperar correctamente el documento original en aproximadamente un 30 % de los casos en el top-5, lo que representa una base sólida para tareas de búsqueda semántica sobre textos breves.

7.3. Testeando el Agente de Sabores

Para este agente se generaron 1000 fórmulas lógicas que representan combinaciones gustativas específicas, incluyendo operadores lógicos como `AND` y `OR`. Las consultas fueron evaluadas contra la base de cócteles mediante lógica difusa, maximizando y minimizando los grados de pertenencia a categorías de sabor.

```
Evaluación del Flavor\_Agent:
Precision@5: 72.22\%
Recall@5: 14.77\%
F1-score: 13.63\%
Cobertura: 77.30\%
```

El sistema alcanza una cobertura considerable, y si bien la recuperación (Recall) es moderada, la precisión en las recomendaciones es alta. Esto demuestra la capacidad del agente para priorizar resultados altamente relevantes, aunque aún se identifican oportunidades para mejorar la recuperación de casos menos evidentes.

7.4. Testeando Metaheurísticas

Durante el proceso de validación de candidatos, se compararon dos enfoques metaheurísticos: **Ant Colony Optimization (ACO)** y **Búsqueda Tabú**. Inicialmente se probó ACO, pero los resultados fueron insatisfactorios. La sustitución por Búsqueda Tabú mostró una mejora significativa en la calidad de las soluciones seleccionadas.

```
Resumen:
 Tabu ganó en 500 casos
 ACO ganó en 0 casos
 Empates: 0
 Promedio ACO:   230.25
 Promedio Tabu:  298.06
```

La búsqueda tabú resultó superior en todos los casos evaluados, tanto en cantidad de victorias como en calidad media de las soluciones. Esta elección refuerza la necesidad de emplear técnicas metaheurísticas robustas que se adapten bien al espacio de soluciones del problema planteado.

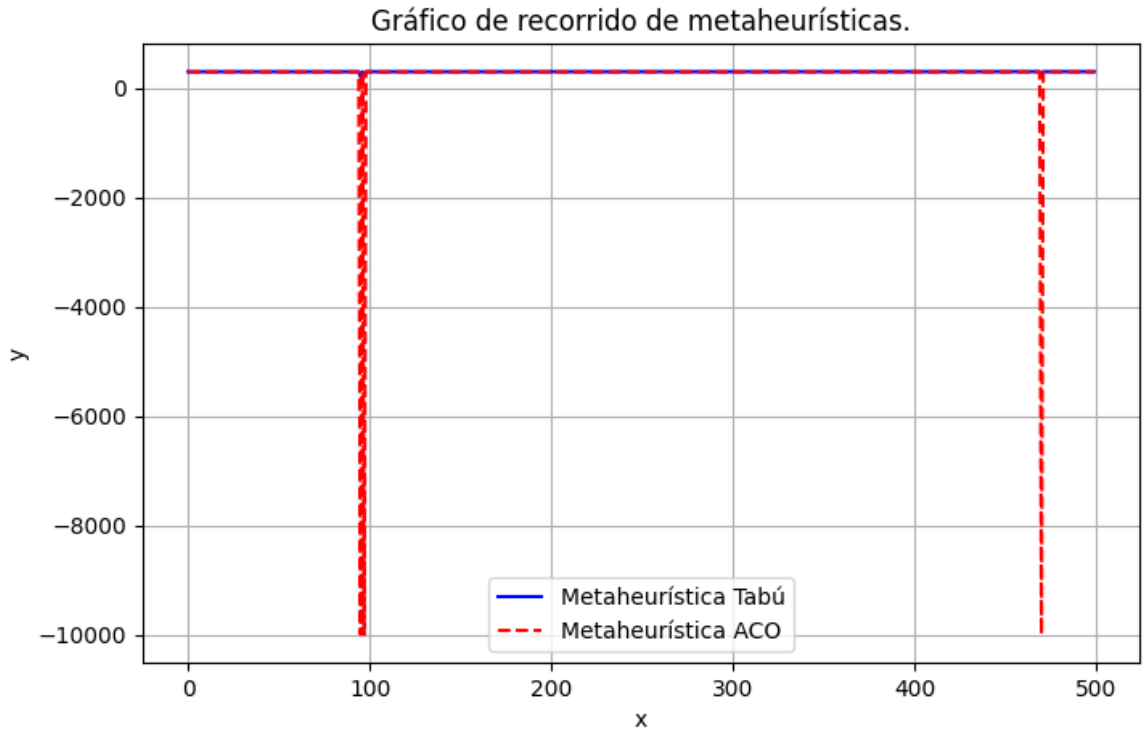

Figura 3. Comparación de rendimiento entre ACO y Búsqueda Tabú.

8. Logros, insuficiencias, y propuestas de mejora:

8.1. Logros Alcanzados:

Durante el desarrollo del sistema, se alcanzaron varios objetivos técnicos y conceptuales relevantes:

- Se implementó un flujo funcional completo, capaz de analizar consultas del usuario y generar respuestas coherentes y contextualizadas.
- Se integraron técnicas metaheurísticas para la selección óptima de respuestas candidatas, considerando restricciones explícitas e implícitas establecidas en las consultas.
- Se aplicaron fundamentos de lógica difusa para evaluar grados de pertenencia entre preferencias de sabor y perfiles de cócteles, complementado con la generación de variables aleatorias basadas en distribuciones probabilísticas derivadas del modelo *softmax*.
- Se construyó un sistema robusto de Recuperación de Información, incorporando embeddings semánticos, una ontología formal del dominio y una arquitectura RAG (Retrieval-Augmented Generation) completamente funcional.

8.2. Limitaciones Identificadas y Propuestas de Mejora:

Durante el desarrollo del sistema, se detectaron diversas insuficiencias que abren oportunidades claras de mejora:

1. **Limitado razonamiento en el Agente de Sabores:** El agente carece de un modelo inferencial profundo sobre la interacción entre sabores. Por ejemplo, no puede deducir que un cóctel amargo con adición de sal podría disminuir la percepción del amargor. Este tipo de razonamiento sensorial-combinatorio está ausente.

Propuesta: Enriquecer el agente de sabores mediante la incorporación de reglas de inferencia específicas. Por ejemplo:

SI (amargo OR muy_amargo) AND poco_salado ENTONCES nada_amargo

Estas reglas pueden ser definidas manualmente a partir de conocimiento experto o aprendidas a partir de datos sensoriales etiquetados.

2. **Desalineación entre la Ontología y las Fuentes Web:** La ontología fue diseñada con base en la estructura de recetas de *Difford's Guide*. Sin embargo, el agente *crawler* puede obtener recetas desde otras fuentes (e.g., *Liquor.com*, *PunchDrink*), que presentan formatos distintos, impidiendo su integración directa a la ontología, aunque sí es posible añadirlas al sistema de embeddings.

   Propuesta: Para cada fuente adicional admitida, se debe identificar y mapear su estructura HTML. Esto permitirá extraer información compatible con la ontología. Se deberá contemplar la posibilidad de campos vacíos cuando ciertos atributos no estén presentes en esa fuente.
3. **Ausencia de pruebas integrales del flujo completo:** No se realizaron pruebas sistemáticas del sistema completo debido a limitaciones de uso diario del modelo de lenguaje (por restricción de tokens). Esto impidió evaluar con precisión la robustez del flujo bajo múltiples escenarios de consulta.

   Propuesta: Programar la ejecución diaria de n simulaciones de consultas, evaluando el desempeño sobre un conjunto de k documentos. Se recomienda monitorear métricas como cobertura de recuperación y precisión semántica, observando su evolución temporal.
4. **Dependencia de un único modelo de lenguaje gratuito:** El sistema ha sido evaluado únicamente con un modelo gratuito. No se ha determinado si existen modelos equivalentes o superiores en desempeño.

   Propuesta: Ejecutar pruebas comparativas utilizando múltiples modelos de lenguaje (gratuitos y, si es posible, comerciales), con el objetivo de analizar sus capacidades en tareas de recuperación, razonamiento y generación de respuestas.

9. Bibliografía:

- Zhao, P., Zhang, H., Yu, Q., Wang, Z., Geng, Y., Fu, F., Yang, L., Zhang, W., Jiang, J., & Cui, B. (2024). *Retrieval-Augmented Generation for AI-Generated Content: A Survey*.
- Jurafsky, D., & Martin, J. H. (2025). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*.
- Oliva Santos, R. *Socialización de la Información y el Conocimiento en la Web. Revista Ciencias Matemáticas*.
- Baeza-Yates, R., Ribeiro-Neto, B., Mills, D., Bonn, O., Juan, S., Milan, M., Taipei, C., & Addison Wesley Longman Limited. (1999). *Modern Information Retrieval*.
- Klir, G. J., & Yuan, B. (1995). *Fuzzy Sets and Fuzzy Logic: Theory and Applications*.
- Alonso, S., Cordón, O., Fernández, I., & Herrera, F. (2004). *La metaheurística de optimización basada en colonias de hormigas: modelos y nuevos enfoques*.
- García Sánchez, Á. (2013). *Técnicas metaheurísticas*.