RAG y Bases de Datos Vectoriales

Retrieval-Augmented Generation (RAG)

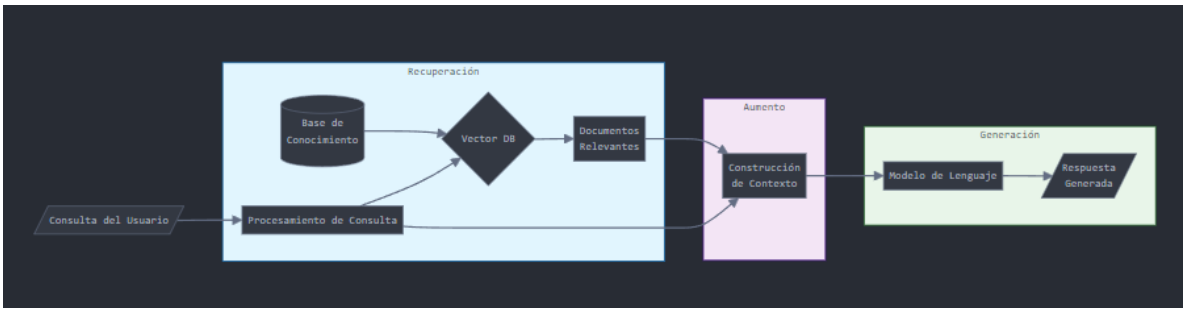
¿Qué es RAG?

Retrieval-Augmented Generation (RAG) es una técnica que combina la capacidad de generación de texto de los modelos de lenguaje grandes (LLMs) con la recuperación de información de fuentes externas. Esta aproximación permite mejorar la precisión y fiabilidad de las respuestas generadas por los modelos de IA.

¿Cómo funciona?

El proceso de RAG se puede dividir en tres pasos principales:

1. **Recuperación (Retrieval)**:
    - Cuando se recibe una consulta, el sistema busca en una base de datos o fuente de conocimiento externa información relevante.
    - Esta información puede estar almacenada en forma de documentos, fragmentos de texto, o datos estructurados.
    - Se utilizan técnicas de búsqueda semántica para encontrar el contenido más pertinente.
2. **Aumento (Augmentation)**:
    - La información recuperada se combina con la consulta original.
    - Se prepara un contexto enriquecido que incluye tanto la pregunta como la información relevante encontrada.
3. **Generación**:
    - El modelo de lenguaje utiliza el contexto aumentado para generar una respuesta.
    - La respuesta se basa tanto en el conocimiento general del modelo como en la información específica recuperada.



Ventajas de RAG

Precisión Mejorada

- Las respuestas están respaldadas por información actualizada y específica.
- Reduce las "alucinaciones" o información incorrecta generada por el modelo.

Conocimiento Actualizable

- La base de conocimiento externa puede actualizarse sin necesidad de reentrenar el modelo.
- Permite mantener la información siempre al día.

Transparencia

- Las fuentes de información pueden ser rastreadas y citadas.
- Mejora la confiabilidad y verificabilidad de las respuestas.

Aplicaciones Comunes

1. **Asistentes Virtuales**
    - Sistemas de atención al cliente con acceso a documentación específica.
    - Asistentes de investigación que pueden consultar bases de datos académicas.
2. **Sistemas de Documentación**
    - Herramientas de búsqueda y consulta de documentación técnica.
    - Sistemas de gestión del conocimiento empresarial.
3. **Herramientas de Investigación**
    - Análisis de documentos legales y normativas.
    - Investigación científica y médica.

Consideraciones de Implementación

Calidad de los Datos

- La calidad de las respuestas depende directamente de la calidad de la información almacenada.
- Es crucial mantener una base de conocimiento precisa y actualizada.

Rendimiento

- El proceso de recuperación debe ser eficiente para mantener tiempos de respuesta aceptables.
- Se requiere una buena estrategia de indexación y búsqueda.

Costos

- La implementación puede requerir recursos significativos de almacenamiento y procesamiento.
- Es importante considerar el equilibrio entre precisión y eficiencia.

Bases de Datos Vectoriales y Embeddings

¿Qué es una Base de Datos Vectorial?

Una base de datos vectorial es un sistema especializado diseñado para almacenar y buscar eficientemente vectores de alta dimensionalidad. Estos vectores son representaciones numéricas (embeddings) de datos como texto, imágenes o audio en un espacio matemático multidimensional.

Características Principales de las Bases de Datos Vectoriales

Las bases de datos vectoriales se distinguen por dos características fundamentales: su estructura de datos única y su capacidad de búsqueda por similitud. Estas características las hacen especialmente adecuadas para aplicaciones modernas de inteligencia artificial y procesamiento de datos.

La estructura de datos en estas bases de datos es fundamentalmente diferente de las bases de datos tradicionales. En lugar de almacenar datos en formatos convencionales como texto o números individuales, las bases de datos vectoriales trabajan con vectores numéricos de longitud fija. Estos vectores son representaciones matemáticas precisas donde cada elemento tiene el mismo número de dimensiones, típicamente oscilando entre 100 y más de 1000 dimensiones. Esta estructura uniforme no es arbitraria; está específicamente diseñada para optimizar las búsquedas de similitud en espacios multidimensionales, permitiendo

operaciones eficientes que serían extremadamente complejas o imposibles con estructuras de datos tradicionales.

La búsqueda por similitud representa otra característica distintiva y crucial de estas bases de datos. A diferencia de las búsquedas exactas en bases de datos tradicionales, las bases de datos vectoriales emplean algoritmos de búsqueda aproximada de vecinos más cercanos (ANN, por sus siglas en inglés). Estos algoritmos están diseñados para encontrar elementos similares basándose en la distancia entre vectores en el espacio multidimensional. Para calcular estas distancias, se utilizan varios métodos matemáticos, cada uno con sus propias ventajas según el caso de uso. La distancia coseno, por ejemplo, es excelente para medir la similitud independientemente de la magnitud de los vectores, mientras que la distancia euclidiana es útil cuando la magnitud también es relevante. El producto punto, por su parte, ofrece otra perspectiva de similitud que puede ser más apropiada en ciertos contextos, como en el procesamiento de lenguaje natural.

Estas características principales no operan de manera aislada, sino que trabajan en conjunto para proporcionar un sistema de almacenamiento y recuperación de datos altamente eficiente. La estructura de datos vectorial facilita la aplicación de los algoritmos de búsqueda por similitud, mientras que los diferentes métodos de cálculo de distancia permiten adaptar el sistema a las necesidades específicas de cada aplicación. Esta combinación hace que las bases de datos vectoriales sean particularmente eficaces en aplicaciones que requieren búsquedas semánticas, recomendaciones personalizadas o cualquier otro caso donde la similitud conceptual sea más importante que la coincidencia exacta.

1. Estructura de Datos

- Almacena datos como vectores numéricos de longitud fija
- Cada vector típicamente tiene entre 100 y 1000+ dimensiones
- Optimizada para búsquedas de similitud en espacios multidimensionales

2. Búsqueda por Similitud

- Utiliza algoritmos de búsqueda aproximada de vecinos más cercanos (ANN)
- Permite encontrar elementos similares basándose en la distancia entre vectores
- Métodos comunes incluyen:
    - Distancia coseno
    - Distancia euclidiana
    - Producto punto

¿Por qué se utilizan las Bases de Datos Vectoriales para Embeddings?

Las bases de datos vectoriales se han convertido en una herramienta fundamental para el manejo de embeddings por tres razones principales: su eficiencia en la búsqueda, su capacidad de representación semántica y su escalabilidad.

La eficiencia en la búsqueda es una de las características más destacadas de estas bases de datos. Los embeddings realizan una tarea crucial al transformar datos complejos, como texto, imágenes o audio, en vectores numéricos que pueden ser procesados eficientemente por computadoras. Las bases de datos vectoriales están específicamente diseñadas para trabajar con estos vectores en espacios multidimensionales, lo que las hace extraordinariamente eficientes para realizar búsquedas. Esta capacidad permite ejecutar búsquedas semánticas rápidas incluso en conjuntos de datos masivos, algo que sería prácticamente imposible con bases de datos tradicionales.

La representación semántica constituye otra ventaja fundamental de este enfoque. Los embeddings tienen la notable capacidad de capturar no solo el contenido literal de los datos, sino también su significado y las relaciones semánticas entre diferentes elementos. Cuando dos elementos son conceptualmente similares, sus embeddings también lo son en el espacio vectorial. Esta característica permite realizar búsquedas conceptuales sofisticadas, superando las limitaciones de las búsquedas por coincidencia exacta. Por ejemplo, una búsqueda de "automóvil" podría encontrar documentos relevantes que contengan palabras como "coche", "vehículo" o "carro", incluso si el término exacto no está presente.

En cuanto a la escalabilidad, las bases de datos vectoriales demuestran un rendimiento excepcional. Están diseñadas desde su concepción para manejar eficientemente millones de vectores, implementando técnicas de indexación altamente especializadas que permiten mantener tiempos de respuesta rápidos incluso cuando la cantidad de datos crece significativamente. Esta capacidad de escalar sin degradar el rendimiento es crucial en aplicaciones modernas, donde el volumen de datos puede crecer exponencialmente. Las bases de datos vectoriales logran esto mediante estructuras de datos optimizadas y algoritmos de búsqueda avanzados que mantienen la eficiencia incluso con conjuntos de datos masivos.

La combinación de estas tres características hace que las bases de datos vectoriales sean la elección ideal para trabajar con embeddings en aplicaciones de inteligencia artificial y procesamiento de lenguaje natural. Su capacidad para manejar eficientemente grandes volúmenes de datos, mientras mantienen la precisión semántica y la velocidad de búsqueda, las convierte en una herramienta indispensable en la infraestructura moderna de procesamiento de datos.

1. Eficiencia en la Búsqueda

- Los embeddings convierten datos complejos en vectores numéricos
- Las bases de datos vectoriales están optimizadas para buscar eficientemente en espacios vectoriales
- Permiten búsquedas semánticas rápidas en grandes conjuntos de datos

2. Representación Semántica

- Los embeddings capturan significado y relaciones semánticas
- Elementos similares tienen vectores similares
- Permite búsquedas conceptuales y no solo por coincidencia exacta

3. Escalabilidad

- Diseñadas para manejar millones de vectores
- Implementan técnicas de indexación especializadas
- Mantienen tiempos de respuesta rápidos incluso con grandes volúmenes de datos

Aplicaciones Comunes

1. Búsqueda Semántica

- Encontrar documentos similares
- Búsqueda de imágenes similares
- Recomendaciones de contenido

2. Sistemas RAG

- Almacenamiento de embeddings de documentos
- Recuperación eficiente de contexto relevante
- Búsqueda de información similar a la consulta

3. Sistemas de Recomendación

- Productos similares
- Contenido relacionado
- Perfiles de usuario similares

Ejemplos de Bases de Datos Vectoriales

1. **Populares de Código Abierto**
    - Faiss (Facebook AI Similarity Search)
    - Milvus
    - Weaviate
    - Qdrant
2. **Servicios Comerciales**
    - Pinecone
    - ChromaDB
    - Redis con módulo de búsqueda vectorial
    - PostgreSQL con pgvector

Consideraciones Técnicas en Bases de Datos Vectoriales

Al implementar y mantener una base de datos vectorial, existen varias consideraciones técnicas cruciales que debemos tener en cuenta para garantizar su óptimo funcionamiento. Estas consideraciones se centran principalmente en tres aspectos fundamentales: la indexación, el rendimiento y el mantenimiento del sistema.

La indexación representa uno de los pilares fundamentales en el funcionamiento de una base de datos vectorial. Para realizar búsquedas eficientes en espacios multidimensionales, estas bases de datos implementan índices especializados que permiten organizar y acceder a los vectores de manera óptima. Una de las técnicas más utilizadas es el Locality-Sensitive Hashing (LSH), que permite agrupar vectores similares en los mismos "cubos" o espacios de almacenamiento, facilitando así su posterior recuperación. Además, se emplean estructuras de datos como los árboles KD y los árboles Ball, que dividen el espacio vectorial en regiones más pequeñas y manejables, permitiendo realizar búsquedas más eficientes al reducir el número de comparaciones necesarias.

El rendimiento de una base de datos vectorial requiere un delicado equilibrio entre la precisión de los resultados y la velocidad de respuesta. Este balance es particularmente importante en aplicaciones que requieren respuestas en tiempo real. La optimización de índices juega un papel crucial en este aspecto, ya que un índice bien diseñado puede reducir significativamente el tiempo de búsqueda sin comprometer la calidad de los resultados. La gestión eficiente de la memoria es otro factor crítico, especialmente cuando se manejan grandes volúmenes de datos vectoriales. Es necesario implementar estrategias de caché y de paginación que permitan mantener en memoria los datos más frecuentemente accedidos mientras se gestiona de manera eficiente el almacenamiento en disco.

El mantenimiento de una base de datos vectorial requiere una atención constante y sistemática. La actualización de índices debe realizarse de manera programada y eficiente para mantener el rendimiento óptimo del sistema, especialmente cuando se agregan o eliminan vectores con frecuencia. La gestión de la dimensionalidad es un aspecto particularmente desafiante, ya que trabajar con vectores de alta dimensionalidad puede llevar a problemas conocidos como la "maldición de la dimensionalidad", donde la eficacia de los algoritmos de búsqueda tradicionales se degrada significativamente. Por último, el monitoreo continuo del rendimiento es esencial para identificar y resolver problemas de manera proactiva, asegurando que la base de datos mantenga niveles óptimos de funcionamiento y responda adecuadamente a las necesidades de la aplicación.

Estas consideraciones técnicas no son elementos aislados, sino que forman parte de un sistema interconectado donde cada aspecto influye en los demás. Por ejemplo, una buena estrategia de indexación puede mejorar significativamente el rendimiento, mientras que un mantenimiento adecuado asegura que tanto los índices como el rendimiento se mantengan en niveles óptimos a lo largo del tiempo. La comprensión y gestión adecuada de estos aspectos

técnicos es fundamental para el éxito en la implementación y operación de una base de datos vectorial.

1. Indexación

- Índices especializados para búsqueda eficiente
- Técnicas como LSH (Locality-Sensitive Hashing)
- Árboles de búsqueda (KD-trees, Ball trees)

2. Rendimiento

- Equilibrio entre precisión y velocidad
- Optimización de índices
- Gestión de memoria

3. Mantenimiento

- Actualización de índices
- Gestión de dimensionalidad
- Monitoreo de rendimiento

Configurar demo

Se lee los parámetros de conexión al modelo desde un archivo config.json que ponemos en la carpeta de la apliacción. Se asignan a los componentes TAiOpenChat y TAiEmbeddings.
El componente TAiRagChat conecta TAiOpenChat y TAiDataVec.

El componente TAiDataVec conecta con TAiEmbeddings.

Funcionamiento RagDemo.dpr

Con el RAG buscamos ofrecer a la IA un contexto con datos propios de forma que obtengamos respuestas más precisas. Si al abrir el programa vamos a la segunda pestaña de Consultas RAG y escribimos una consulta sobre unos datos específicos nuestros, lo más normal es que la IA no sepa qué responder o de resultados inexactos. El contexto de RAG lo podemos proporcionar cargando un texto plano o un JSON, al cargarlo se hace una petición a OpenAI que genera los vectores de estos archivos. Estos vectores los podemos guardar en memoria o en una base de datos preparada para hacerlo, por ejemplo, Postgres con la extensión PgVector.

Cuando hacemos una consulta a OpenAi, realizamos una consulta previa a la base de datos de vectores para obtener los vectores más próximos a nuestro prompt y que añadimos a la consulta como contexto.

Esta es una librería en Delphi que implementa funcionalidad para RAG (Retrieval-Augmented Generation) utilizando embeddings de texto. Los componentes principales son:

1. `TAiEmbeddingNode`: Una clase que representa un embedding individual, incluyendo:

- Los datos del vector embedding
- El texto original
- Funcionalidad para calcular similitud por coseno entre embeddings
- Conversión desde/hacia JSON

2. `TAiDataVec`: Un contenedor principal para almacenar y gestionar colecciones de embeddings que:

- Permite añadir embeddings desde texto plano o JSON
- Guarda/carga embeddings desde archivos
- Mantiene un índice para búsquedas
- Permite búsqueda de embeddings similares

3. `TAIEmbeddingIndex`: Una clase base para implementar índices de búsqueda, con:

- Una implementación básica (`TAIBasicEmbeddingIndex`) que hace búsqueda por similitud de coseno
- Soporte para límites de resultados y umbral de precisión
- Posibilidad de implementar índices más sofisticados

4. `TAiRagChat`: Un componente que integra:

- La funcionalidad de búsqueda de embeddings
- Un componente de chat (presumiblemente para LLMs)
- Capacidad de realizar consultas RAG combinando el contexto encontrado con la pregunta

Es una librería que facilita la implementación de sistemas RAG en aplicaciones Delphi, permitiendo:

1. Crear y gestionar embeddings
2. Buscar información relevante por similitud semántica
3. Integrar esa información con modelos de lenguaje para respuestas mejoradas con contexto

La librería está diseñada para ser extensible y configurable, permitiendo diferentes implementaciones de índices y métodos de búsqueda.

Manual de Usuario para la Librería de Componentes RAG en Delphi

Introducción

El presente manual tiene como objetivo proporcionar una guía completa para la implementación y uso de la librería de componentes RAG (Retrieval-Augmented Generation) desarrollada en Delphi. Esta librería integra técnicas avanzadas de procesamiento de lenguaje natural (PLN) con funcionalidades para manejar embeddings y consultas semánticas, permitiendo crear sistemas inteligentes basados en inteligencia artificial.

¿Qué es RAG?

Retrieval-Augmented Generation (RAG) es un enfoque que combina capacidades de generación de texto de modelos de lenguaje grandes (LLMs) con la recuperación de información relevante desde fuentes externas. Este enfoque es ideal para aplicaciones donde la precisión y el contexto actualizado son esenciales, como asistentes virtuales, sistemas de recomendación y motores de búsqueda semántica.

Características de la Librería RAG

Esta librería está diseñada específicamente para desarrolladores que trabajan en Delphi y necesitan integrar RAG en sus aplicaciones. Entre sus principales características se incluyen:

- Creación y gestión de embeddings, que son representaciones vectoriales de datos.
- Búsqueda de información relevante utilizando índices vectoriales.
- Capacidad para enriquecer consultas mediante recuperación de contexto semántico.
- Compatibilidad con bases de datos vectoriales populares, como PostgreSQL con PgVector.

Beneficios de Usar la Librería RAG

1. **Respuestas Más Precisas**: Al incorporar contexto específico en las consultas, se mejora la calidad de las respuestas generadas.
2. **Flexibilidad y Escalabilidad**: Permite manejar grandes volúmenes de datos y adaptarse a diferentes requisitos de indexación y búsqueda.
3. **Extensibilidad**: Soporta personalización mediante la implementación de índices avanzados o modificaciones en los componentes.

Estructura del Manual

El manual está organizado en las siguientes secciones:

1. **Componentes Principales de la Librería**: Explicación detallada de las clases y funcionalidades incluidas.
2. **Configuración del Entorno**: Guía para configurar el proyecto Delphi y conectar con bases de datos vectoriales.
3. **Uso de la Librería**: Instrucciones paso a paso para implementar funcionalidades de RAG.
4. **Ejemplo Práctico**: Implementación de un proyecto con la librería para demostrar su funcionalidad.
5. **Casos de Uso**: Aplicaciones comunes y ventajas en escenarios prácticos.
6. **Consideraciones Técnicas**: Recomendaciones para optimizar el rendimiento y gestionar recursos.

¿Quién Debería Usar Este Manual?

Este documento está dirigido a:

- Desarrolladores que utilicen Delphi y busquen integrar funcionalidades de inteligencia artificial en sus aplicaciones.
- Ingenieros interesados en implementar sistemas de recuperación y generación de información.
- Equipos de desarrollo que trabajen en aplicaciones como asistentes virtuales, sistemas de recomendación o motores de búsqueda.

2. Componentes Principales de la Librería

La librería RAG para Delphi está diseñada con una arquitectura modular, facilitando la integración y personalización de sus funcionalidades. En esta sección, se describen los componentes principales, sus propiedades y métodos clave, así como su relación con otros elementos de la librería.

2.1 TAiEmbeddingNode

Descripción:
`TAiEmbeddingNode` representa un nodo de embedding que encapsula datos vectoriales, el texto asociado y funcionalidades para calcular similitudes semánticas entre embeddings. Este componente es esencial para trabajar con representaciones numéricas de texto y realizar búsquedas por similitud.

Propiedades Principales:

- `FData`: Contiene los datos del vector embedding.
- `FText`: Almacena el texto original del cual se generó el embedding.
- `FDim`: Dimensión del vector embedding.
- `FModel`: Nombre del modelo utilizado para generar el embedding.

Métodos Relevantes:

- **`CosineSimilarity(Node: TAiEmbeddingNode): Double`**
  Calcula la similitud coseno entre dos embeddings.
- **`ToJSON: TJSONObject`**
  Convierte un embedding en formato JSON.
- **`FromJSON(JSON: TJSONObject)`**
  Carga un embedding desde un objeto JSON.

Ejemplo de Uso:

```
pascal
Copiar código
var
  Node1, Node2: TAiEmbeddingNode;
  Similarity: Double;
begin
  Node1 := TAiEmbeddingNode.Create;
  Node2 := TAiEmbeddingNode.Create;

  // Asignar datos a los embeddings
  Node1.FData := [0.1, 0.2, 0.3];
  Node2.FData := [0.1, 0.2, 0.4];

  // Calcular similitud coseno
  Similarity := Node1.CosineSimilarity(Node2);
  Writeln('Similitud Coseno: ', Similarity:0:2);
end;
```

2.2 TAiDataVec

Descripción:
`TAiDataVec` actúa como un contenedor para manejar colecciones de embeddings. Este componente permite añadir, almacenar, buscar y recuperar embeddings basados en similitudes.

Propiedades Principales:

- `Embeddings`: Lista de embeddings almacenados.
- `Index`: Índice interno para búsquedas rápidas.

Métodos Relevantes:

- **`AddEmbedding(Text: String)`**
  Genera un embedding para un texto dado y lo añade a la colección.
- **`Search(SimilarityThreshold: Double; MaxResults: Integer): TArray<TAiEmbeddingNode>`**
  Realiza una búsqueda de embeddings similares dentro del índice.
- **`SaveToFile(FileName: String)`**
  Guarda los embeddings en un archivo.

- **`LoadFromFile(FileName: String)`**
  Carga embeddings desde un archivo.

Ejemplo de Uso:

```
pascal
Copiar código
var
  DataVec: TAiDataVec;
  Results: TArray<TAiEmbeddingNode>;
begin
  DataVec := TAiDataVec.Create;

  // Añadir embeddings
  DataVec.AddEmbedding('Texto de ejemplo 1');
  DataVec.AddEmbedding('Texto de ejemplo 2');

  // Buscar embeddings similares
  Results := DataVec.Search(0.8, 5);
  Writeln('Embeddings similares encontrados: ', Length(Results));
end;
```

2.3 TAiEmbeddingIndex

Descripción:
Este componente es responsable de indexar y realizar búsquedas eficientes en colecciones de embeddings. La clase base `TAiEmbeddingIndex` puede extenderse para implementar diferentes estrategias de indexación.

Tipos de Índices Soportados:

- **`TAIBasicEmbeddingIndex`**: Búsqueda basada en similitud coseno.
- **`TAIHNSWIndex`**: Búsqueda eficiente en grandes volúmenes de datos.

Métodos Relevantes:

- **`BuildIndex(Embeddings: TList<TAiEmbeddingNode>)`**
  Construye el índice a partir de una lista de embeddings.
- **`Search(Query: TAiEmbeddingNode; TopK: Integer): TArray<TAiEmbeddingNode>`**
  Devuelve los `TopK` resultados más cercanos a un embedding de consulta.

2.4 TAiRagChat

Descripción:
`TAiRagChat` conecta la funcionalidad de embeddings con modelos de lenguaje (LLMs).

Este componente integra consultas con contexto adicional proporcionado por los embeddings recuperados.

Propiedades Principales:

- `ChatModel`: Interfaz para interactuar con un modelo de lenguaje.
- `DataVec`: Referencia al contenedor de embeddings.

Métodos Relevantes:

- **`QueryWithContext(QueryText: String): String`**
  Realiza una consulta a un LLM añadiendo contexto relevante.

Ejemplo de Uso:

```
var
  RagChat: TAiRagChat;
begin
  RagChat := TAiRagChat.Create;

  // Configurar modelo y embeddings
  RagChat.ChatModel := TAiOpenChat.Create('config.json');
  RagChat.DataVec := TAiDataVec.Create;

  // Realizar una consulta
  Writeln(RagChat.QueryWithContext('¿Cuál es el precio del producto?'));
end;
```

4. Uso de la Librería: Trabajando en Memoria

Los componentes de la librería permiten manejar embeddings directamente en memoria, lo que simplifica la implementación y elimina la dependencia de bases de datos externas. Este enfoque es ideal para escenarios de prototipado rápido o cuando el volumen de datos es pequeño.

4.1 Flujo de Trabajo en Memoria

El flujo de trabajo básico para RAG en memoria es el siguiente:

1. Crear y gestionar embeddings utilizando `TAiEmbeddingNode` y `TAiDataVec`.
2. Realizar búsquedas semánticas utilizando los índices en memoria.
3. Integrar los resultados en consultas RAG mediante `TAiRagChat`.

4.2 Configuración Inicial

Inicialización de Componentes:

```
pascal
Copiar código
var
  DataVec: TAiDataVec;
begin
  // Crear el contenedor de embeddings
  DataVec := TAiDataVec.Create;

  // Configurar embeddings sin base de datos externa
  Writeln('Sistema de RAG inicializado en memoria.');
end;
```

4.3 Crear y Almacenar Embeddings

Los embeddings se pueden generar desde texto plano y almacenar en memoria dentro de un `TAiDataVec`.

Ejemplo: Añadir Embeddings desde Texto

```
pascal
Copiar código
var
  DataVec: TAiDataVec;
begin
  DataVec := TAiDataVec.Create;

  // Añadir embeddings a partir de textos
  DataVec.AddEmbedding('Este es un ejemplo de texto.');
  DataVec.AddEmbedding('Otro texto que será usado como referencia.');

  Writeln('Embeddings generados y almacenados en memoria.');
end;
```

4.4 Búsqueda de Embeddings Similares

Una vez generados los embeddings, se pueden buscar los más similares en memoria.

Ejemplo: Búsqueda Semántica

```
pascal
Copiar código
var
  DataVec: TAiDataVec;
  Results: TArray<TAiEmbeddingNode>;
begin
  DataVec := TAiDataVec.Create;

  // Añadir textos como embeddings
  DataVec.AddEmbedding('Texto 1 de prueba.');
  DataVec.AddEmbedding('Texto 2 de ejemplo.');
```

```
  // Buscar similitud con un texto nuevo
  Results := DataVec.Search('Texto para buscar similitudes.', 0.75, 5);

  // Mostrar resultados
  Writeln('Embeddings similares encontrados: ', Length(Results));
end;
```

4.5 Consultas RAG con Contexto

Con `TAiRagChat`, puedes integrar los resultados de las búsquedas en consultas RAG enriquecidas.

Ejemplo: Consultas RAG sin Base de Datos

```
pascal
Copiar código
var
  RagChat: TAiRagChat;
  DataVec: TAiDataVec;
  Response: String;
begin
  DataVec := TAiDataVec.Create;
  RagChat := TAiRagChat.Create;

  // Configurar componentes
  RagChat.DataVec := DataVec;

  // Añadir embeddings al contenedor en memoria
  DataVec.AddEmbedding('Este es un dato contextual importante.');
  DataVec.AddEmbedding('Otro contexto relevante para el sistema.');

  // Realizar consulta con contexto
  Response := RagChat.QueryWithContext('¿Qué información tienes sobre el
sistema?');
  Writeln('Respuesta de la IA: ', Response);
end;
```

Ventajas del Trabajo en Memoria

1. **Simplicidad**: No requiere configuración adicional ni conexión a bases de datos externas.
2. **Velocidad**: Ideal para consultas rápidas en volúmenes pequeños de datos.
3. **Portabilidad**: El sistema funciona completamente en memoria, lo que lo hace fácil de distribuir y probar.

Limitaciones del Trabajo en Memoria

1. **Escalabilidad**: La memoria RAM limita la cantidad de embeddings que se pueden manejar.
2. **Persistencia**: Los datos no se almacenan entre ejecuciones; es necesario cargarlos nuevamente si se cierran.
3. **Rendimiento en Búsquedas**: Aunque eficiente para volúmenes pequeños, la búsqueda en memoria puede no ser óptima para grandes conjuntos de datos.

```
CREATE TABLE IF NOT EXISTS public.anexos
(
    id BigSerial,
    fechadoc TimeStamp,
    categoria character varying(300),
    pathdoc character varying(300),
    filename character varying(300),
    resumen text,
    json jsonb,
    fileformat character varying(10),
    embedding vector(1536),
    PRIMARY KEY (id)
)
```