User Manual for RAG Components Library in Delphi

Introduction

This manual provides comprehensive guidance for implementing and using the RAG (Retrieval-Augmented Generation) components library developed in Delphi. This library integrates advanced natural language processing (NLP) techniques with embedding and semantic query capabilities, enabling the creation of AI-based intelligent systems.

What is RAG?

Retrieval-Augmented Generation (RAG) is an approach that combines large language models' (LLMs) text generation capabilities with information retrieval from external sources. This approach is ideal for applications where accuracy and up-to-date context are essential, such as virtual assistants, recommendation systems, and semantic search engines.

RAG Library Features

This library is specifically designed for Delphi developers who need to integrate RAG into their applications. Its main features include:

- Creation and management of embeddings (vector representations of data)
- Relevant information search using vector indices
- Query enrichment through semantic context retrieval
- Compatibility with popular vector databases like PostgreSQL with PgVector

Benefits of Using the RAG Library

1. More Accurate Responses: Incorporating specific context in queries improves the quality of generated responses
2. Flexibility and Scalability: Handles large data volumes and adapts to different indexing and search requirements
3. Extensibility: Supports customization through advanced index implementation or component modifications

Manual Structure

The manual is organized into the following sections:

1. Library Main Components: Detailed explanation of included classes and functionalities

2. Environment Setup: Guide for configuring Delphi projects and connecting to vector databases
3. Library Usage: Step-by-step instructions for implementing RAG functionalities
4. Practical Example: Library implementation project demonstrating its functionality
5. Use Cases: Common applications and advantages in practical scenarios
6. Technical Considerations: Recommendations for performance optimization and resource management

Target Audience

This document is intended for:

- Delphi developers seeking to integrate AI functionalities into their applications
- Engineers interested in implementing information retrieval and generation systems
- Development teams working on applications like virtual assistants, recommendation systems, or search engines

2. Library Main Components

The RAG library for Delphi is designed with a modular architecture, facilitating integration and customization of its functionalities. This section describes the main components, their properties and key methods, and their relationship with other library elements.

2.1 TAiEmbeddingNode

Description: TAiEmbeddingNode represents an embedding node that encapsulates vector data, associated text, and functionalities for calculating semantic similarities between embeddings. This component is essential for working with numerical representations of text and performing similarity searches.

Main Properties:

- `FData`: Contains embedding vector data
- `FText`: Stores original text from which the embedding was generated
- `FDim`: Embedding vector dimension
- `FModel`: Name of the model used to generate the embedding

Relevant Methods:

- `CosineSimilarity(Node: TAiEmbeddingNode): Double` Calculates cosine similarity between two embeddings
- `ToJSON: TJSONObject` Converts embedding to JSON format
- `FromJSON(JSON: TJSONObject)` Loads embedding from a JSON object

Usage Example:

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

  // Assign data to embeddings
  Node1.FData := [0.1, 0.2, 0.3];
  Node2.FData := [0.1, 0.2, 0.4];

  // Calculate cosine similarity
  Similarity := Node1.CosineSimilarity(Node2);
  Writeln('Cosine Similarity: ', Similarity:0:2);
end;
```

2.2 TAiDataVec

Description: TAiDataVec acts as a container for managing embedding collections. This component allows adding, storing, searching, and retrieving embeddings based on similarities.

Main Properties:

- `Embeddings`: List of stored embeddings
- `Index`: Internal index for quick searches

Relevant Methods:

- `AddEmbedding(Text: String)` Generates an embedding for given text and adds it to the collection
- `Search(SimilarityThreshold: Double; MaxResults: Integer): TArray<TAiEmbeddingNode>` Performs similar embeddings search within the index
- `SaveToFile(FileName: String)` Saves embeddings to a file
- `LoadFromFile(FileName: String)` Loads embeddings from a file

Usage Example:

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

  // Add embeddings
  DataVec.AddEmbedding('Example text 1');
  DataVec.AddEmbedding('Example text 2');

  // Search similar embeddings
  Results := DataVec.Search(0.8, 5);
  Writeln('Similar embeddings found: ', Length(Results));
end;
```

2.3 TAiEmbeddingIndex

Description: This component is responsible for indexing and performing efficient searches in embedding collections. The base class TAiEmbeddingIndex can be extended to implement different indexing strategies.

Supported Index Types:

- `TAIBasicEmbeddingIndex`: Cosine similarity-based search
- `TAIHNSWIndex`: Efficient search in large data volumes

Relevant Methods:

- `BuildIndex(Embeddings: TList<TAiEmbeddingNode>)` Builds index from an embeddings list
- `Search(Query: TAiEmbeddingNode; TopK: Integer): TArray<TAiEmbeddingNode>` Returns TopK closest results to a query embedding

2.4 TAiRagChat

Description: TAiRagChat connects embedding functionality with language models (LLMs). This component integrates queries with additional context provided by retrieved embeddings.

Main Properties:

- `ChatModel`: Interface for interacting with a language model
- `DataVec`: Reference to the embeddings container

Relevant Methods:

- `QueryWithContext(QueryText: String): String` Performs an LLM query adding relevant context

Usage Example:

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

  // Configure model and embeddings
  RagChat.ChatModel := TAiOpenChat.Create('config.json');
  RagChat.DataVec := TAiDataVec.Create;

  // Perform a query
  Writeln(RagChat.QueryWithContext('What is the product price?'));
end;
```

4. Library Usage: Working in Memory

The library components allow handling embeddings directly in memory, simplifying implementation and eliminating external database dependencies. This approach is ideal for rapid prototyping scenarios or when dealing with small data volumes.

4.1 In-Memory Workflow

The basic workflow for RAG in memory follows these steps:

1. Create and manage embeddings using TAiEmbeddingNode and TAiDataVec
2. Perform semantic searches using in-memory indices
3. Integrate results in RAG queries through TAiRagChat

4.2 Initial Setup

Component Initialization:

```
var
  DataVec: TAiDataVec;
begin
  // Create embeddings container
  DataVec := TAiDataVec.Create;

  // Configure embeddings without external database
  Writeln('RAG system initialized in memory.');
end;
```

4.3 Creating and Storing Embeddings

Embeddings can be generated from plain text and stored in memory within a TAiDataVec.

Example: Adding Embeddings from Text

```
var
  DataVec: TAiDataVec;
begin
  DataVec := TAiDataVec.Create;

  // Add embeddings from texts
  DataVec.AddEmbedding('This is an example text.');
  DataVec.AddEmbedding('Another text that will be used as reference.');
  Writeln('Embeddings generated and stored in memory.');
end;
```

4.4 Similar Embeddings Search

Once embeddings are generated, you can search for the most similar ones in memory.

Example: Semantic Search

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

  // Add texts as embeddings
  DataVec.AddEmbedding('Test text 1.');
  DataVec.AddEmbedding('Example text 2.');

  // Search similarity with new text
  Results := DataVec.Search('Text to search similarities.', 0.75, 5);

  // Show results
  Writeln('Similar embeddings found: ', Length(Results));
end;
```

4.5 RAG Queries with Context

With TAiRagChat, you can integrate search results into enriched RAG queries.

Example: RAG Queries without Database

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

  // Configure components
  RagChat.DataVec := DataVec;

  // Add embeddings to in-memory container
  DataVec.AddEmbedding('This is important contextual data.');
  DataVec.AddEmbedding('Another relevant context for the system.');

  // Perform query with context
  Response := RagChat.QueryWithContext('What information do you have
about the system?');
  Writeln('AI Response: ', Response);
end;
```

In-Memory Work Advantages

1. Simplicity: No additional configuration or external database connection required
2. Speed: Ideal for quick queries with small data volumes
3. Portability: System works entirely in memory, making it easy to distribute and test

In-Memory Work Limitations

1. Scalability: RAM memory limits the number of manageable embeddings
2. Persistence: Data isn't stored between executions; it needs to be reloaded when closed
3. Search Performance: Though efficient for small volumes, in-memory search may not be optimal for large datasets

5. Database Integration with Vector Storage

5.1 Overview

The library components can work with vector databases by implementing two key events: `OnDataVecAddItem` and `OnDataVecSearch`. This enables RAG implementation over a database system like PostgreSQL with pg_vector extension.

5.2 Requirements

- PostgreSQL database with pg_vector extension installed
- FireDAC components for database connectivity
- Table structure with vector storage capabilities

5.3 Database Implementation

Creating the Database Table

```
CREATE TABLE RagDemo (
    id SERIAL PRIMARY KEY,
    categoria VARCHAR(100),
    texto TEXT,
    embedding vector(1536)
);
```

Implementing Event Handlers

OnDataVecAddItem Event: This event handles storing embeddings in the database.

```
procedure TForm69.DataVec1DataVecAddItem(Sender: TObject;
  aItem: TAiEmbeddingNode; var Handled: Boolean);
var
  Query: TFDQuery;
  sEmbedding, Texto: String;
  JArr: TJSonArray;
begin
  if ChBDMemoria.IsChecked then
  begin
    Handled := False;
    Exit;
  end;

  Query := NewQuery(DbConn, '');
  try
    JArr := aItem.ToJsonArray;
    try
      sEmbedding := JArr.ToString;
    finally
      JArr.Free;
    end;

    Query.SQL.Clear;
    Query.SQL.Add('Insert into RagDemo(categoria, texto, embedding)');
    Query.SQL.Add('VALUES (:categoria, :texto, :embedding)');
    Query.SQL.Add('Returning Id');
    Query.Params.ParamByName('categoria').AsString := 'motofacil';
    Query.Params.ParamByName('texto').AsString := aItem.Text;
    Query.Params.ParamByName('embedding').AsString := sEmbedding;
    Query.Open;
    Handled := True;
  finally
    Query.Free;
  end;
end;
```

OnDataVecSearch Event: This event performs similarity searches using vector operations in PostgreSQL.

```
procedure TForm69.DataVec1DataVecSearch(Sender: TObject;
  Target: TAiEmbeddingNode; aLimit: Integer; aPrecision: Double;
  var aDataVec: TAiDataVec; var Handled: Boolean);
var
  Query: TFDQuery;
  sEmbedding: String;
  JArr: TJSonArray;
  Emb: TAiEmbeddingNode;
begin
  if ChBDMemoria.IsChecked then
  begin
```

```
    Handled := False;
    Exit;
  end;

  aDataVec := TAiDataVec.Create(nil);
  Query := NewQuery(DbConn, '');
  try
    JArr := Target.ToJsonArray;
    try
      sEmbedding := JArr.ToString;
    finally
      JArr.Free;
    end;

    Query.SQL.Clear;
    Query.SQL.Add('SELECT id, texto, embedding <-> :embedding as 
distance');
    Query.SQL.Add('FROM RagDemo');
    Query.SQL.Add('ORDER BY embedding <-> :embedding');
    Query.SQL.Add('LIMIT :limit');
    Query.Params.ParamByName('embedding').AsString := sEmbedding;
    Query.Params.ParamByName('limit').AsInteger := aLimit;
    Query.Open;

    while not Query.Eof do
    begin
      Emb := TAiEmbeddingNode.Create(Target.Dim);
      Emb.Text := Query.FieldByName('texto').AsString;
      aDataVec.AddItem(Emb);
      Query.Next;
    end;
    Handled := True;
  finally
    Query.Free;
  end;
end;
```

5.4 Switching Between Memory and Database Storage

The implementation includes a toggle mechanism (`ChBDMemoria.IsChecked`) to switch between in-memory and database storage:

- When `Handled = False`, the system uses default in-memory storage
- When `Handled = True`, the system uses the database implementation

5.5 Key Benefits of Database Implementation

1. Scalability for large datasets
2. Persistence across application restarts
3. Efficient vector similarity search using pg_vector

4. Support for concurrent access and multiple users