Implementation

Data collection and preprocessing

The first step in the methodology involves collecting and preprocessing financial data from diverse sources to build a comprehensive knowledge base. This data encompasses various financial concepts, products, regulations, and market trends relevant to the advisory domain. Currently there are several datasets available for financial data, such as structured reports, market data feeds, regulatory documents. However for this particular use case we will not focus on such data, as it may cause noise among the relevant information required for the dataset. Instead, we will focus on creating a custom dataset that is tailored to the financial advisory domain.

To achieve that goal our primary tool is web scraping, which allows us to extract information from financial websites, blogs, forums, and regulatory bodies. In this particular case we will focus on extracting information from financial blogs, as they provide a rich source of up-to-date and relevant content, taking particular interest in the website https://www.investopedia.com/. This website is a well-known financial education platform that covers a wide range of topics, including investing, trading, personal finance, and economics. By scraping content from Investopedia, we can gather a diverse set of financial concepts, definitions, and explanations that are commonly sought after by users seeking financial advice.

Rationale: Web scraping from Investopedia ensures that the data is current, relevant, and comprehensive, which is crucial for providing accurate financial advice. This approach also allows for the creation of a custom dataset that is specifically tailored to the needs of the financial advisory domain.

The extracted data is then preprocessed to remove irrelevant content, standardize formats, and annotate entities and relationships. This preprocessing step involves natural language processing techniques such as tokenization, lemmatization, and entity recognition to ensure the data is structured and semantically enriched. For this process we will use NLTK, which is an all purpose NLP Python library. The main purpose of this step is to create a clean and structured dataset that can be effectively utilized for graph construction and information retrieval.

Rationale: Using NLTK for preprocessing ensures that the data is clean, structured, and semantically enriched, which is essential for accurate graph construction and information retrieval.

Graph construction

The core of the methodology lies in constructing a graph-based knowledge representation that captures the intricate relationships between financial concepts. This graph is built using the preprocessed data, where each financial concept is represented as a node and relationships between concepts are represented as edges. The graph structure enables efficient traversal, context-aware retrieval, and semantic understanding of the financial domain.

This step was not manually done, but instead using a very useful tool called Langchain, particularly LangGraph, which is a collection of tools for building and querying knowledge graphs from text data. LangGraph uses state-of-the-art natural language processing models to extract entities, relationships, and hierarchies from unstructured text, and then constructs a graph database that can be queried for information retrieval. By leveraging LangGraph, we can automate the graph construction process and ensure the scalability and accuracy of the resulting knowledge graph. It's worth noting that Langchain is a set of tools that work with third party LLM models, but not provides them as default, meaning you as an user need to provide API keys for the models you want to use. I personally used the GPT-4o-mini model, which is an all purpose and relatively fast model made by OpenAI for this task.

Rationale: Automating the graph construction process with Langchain ensures scalability and accuracy, while using the GPT-4o-mini model provides efficient and accurate entity extraction and relationship mapping.

The way it works is:

1. Provide a clean set of documents (in this case, financial blog articles from Investopedia post-preprocessing).
2. Pass those documents through LangGraph's pipeline, using either a `Tool` or a `Prompt` approach. The tool approach uses the underlying structured output approach of the model, making it output the response in json format with the nodes and edges of the graph. The prompt approach is semantically the same but differs sometimes in the output, instead of using the structured output, it uses a prompt to model the response of the system, a prompt that can be customized by the user.
3. After that processing the system collects all the entities found in all of the documents as well as the entities between them. This is particularly useful as it allows the system to create relationships between entities that are not directly related in the same document, and also uses the power of a language model to create a more accurate and complete graph.
4. Then we can construct the graph using a graph database, in this case we will use Neo4j, which is a popular graph database that allows for efficient storage, querying, and visualization of graph data. The entities are stored as nodes in the graph, and the relationships are stored as edges, forming a connected and structured representation of the financial knowledge base.

Rationale: Using Neo4j for graph storage and querying ensures efficient and scalable management of the knowledge graph, while the structured output approach of LangGraph ensures accurate and comprehensive graph construction.

The resulting graph is a textual graph, meaning that the relationships and nodes are pure text, no extra properties are added to them.

Vector embedding

To enable efficient retrieval and generation of financial advice, the methodology incorporates vector embedding techniques to represent textual data in a continuous vector space. This step is crucial for calculating semantic similarities between user queries and graph nodes, as well as generating contextually relevant responses using language models.

For this task we will use a Hugging Face model in the form of a Sentence Transformer, which is a transformer-based model that maps sentences to high-dimensional vectors in a semantic space. This allows us to encode both user queries and graph nodes as vectors, enabling fast and accurate similarity calculations.

Rationale: Using Sentence Transformers from Hugging Face ensures high accuracy and performance in semantic similarity calculations, which is essential for accurate information retrieval and response generation.

The vector embedding process involves encoding each node and edge in the graph as a vector representation, which is then stored in the a vector database for efficient retrieval (MongoDB) each to its own collection (edge and node). When a user query is received, it is also encoded as a vector using the same Sentence Transformer model, and the most similar nodes and edges in the graph are retrieved based on cosine similarity.

Rationale: Storing vector embeddings in MongoDB ensures efficient retrieval and scalability, while using cosine similarity for matching ensures accurate and relevant information retrieval.

For testing purposes we will also perform a vector embedding to each document as a whole.

Answering a query

The retrieval and generation mechanism of the system is designed to efficiently retrieve relevant financial concepts and relationships based on user queries. This process contains several key steps: query processing, vector similarity calculation, subgraph extraction and answer generation.

As mentioned before the query is cleaned in the same way the documents were, and then passed through the Sentence Transformer model to get a vector representation of the query. Then the system calculates the cosine similarity between the query vector and all the nodes and edges in the graph. The top-k most similar nodes and edges are then selected to form a subgraph that contains the most relevant information to the user query.

Rationale: Using cosine similarity for matching ensures that the most relevant nodes and edges are selected, while forming a subgraph ensures that the retrieved information is contextually relevant and comprehensive.

The subgraph from the resulting retrieval is a broad term, but we have used 2 different approaches to create it, each with its own trade-offs:

1. Basic: this approach is extremely simple as in the resulting subgraph will only include the retrieved nodes and edges and their direct neighbors. This approach is fast and efficient, but may not capture the full context of the query. However, the results are still very good as we will see later
2. Advanced: this approach is more complex and compute expensive. Otherwise known as the Minimum Spanning Subgraph, esentially an optimization problem in which we would like to find the connected subgraph that minimices the amount of edges and nodes involved, while also ensuring that all the relevant nodes and edges are present, if no such subgraph exists, then split the graph in connected components are return a Minimum Spanning Subgraph for each component, of course working with the top relevant nodes and edges that are present in the respective component.

Rationale: Using both basic and advanced approaches for subgraph extraction ensures flexibility and accuracy in capturing the full context of the query, while balancing computational efficiency, depending on which one chooses to use.

Now given the subgraph, we use a language model to generate a response that is contextually relevant to the user query. In this case we will use the Gemini-2.0-exp model, a state of the art model developed by the Google AI team, proficcient in textual tasks. The prompt is constructed by concatenating the user query with the textual representation of the subgraph, which provides the model with the necessary context to generate informative and coherent responses.

Rationale: Using the Gemini-2.0-exp model ensures high-quality and contextually relevant response generation, while constructing the prompt with the subgraph ensures that the model has the necessary context to generate accurate answers.

Using an expert army

Now what we have seen so far is a system that is capable of generating responses based on a user query, but what if the user query is not clear enough or the system is not able to retrieve the necessary information? This is where the expert army comes in.

The expert army is a collection of domain experts built in the same way as the system, but with the different that each one is an expert in a different field of the financial domain. Each so called “expert”, will ideally have a different set of documents, and a different prompt for generating the answer. Making the answer one expert generate, while also relatively similar to another one, with a completely different focus. For example, in this work we are going to use 3 different experts:

personal finance, economics and investment. Each one will have a designated database made from articles and blogs from its field.

Rationale: Using an expert army ensures that the system can handle a wide range of financial queries with high accuracy and depth, while leveraging domain-specific expertise for more accurate and comprehensive answers.

Then we will ask each expert to generate an answer based on the query provided and we will have a agent orchestrator that will gather those answers and in another LLM call combine them in a single answer for the user. This results in a multifaceted answer, relying heavily on which experts we chose for the task and how we comprise the dataset for each of them.

Rationale: Combining answers from multiple experts ensures that the final response is multifaceted and comprehensive, leveraging the strengths of each domain-specific expert for a more accurate and informative answer.