Evaluation Event II - *fierce-blade*

1st Adam Bauer (*adam.bauer*) 2nd Mete Polat (*mete.polat*)

October 28, 2024

1 Introduction

Our current agent can answer factual questions that include one relation and one movie entity. If the answer is not found in the graph, our agent uses embeddings to compute a response. This enables the agent to easily answer questions like "Who is the screenwriter of The Masked Gang: Cyprus?" or "When was 'The Godfather' released?" If the answer cannot be found or computed, or if the entity or relation is not recognized, the agent politely asks the user to rephrase the question or to ask something else.

2 Capabilities

As shown in Figure 1, the agent is divided into five main components: the Main Application, Answering Service, Extraction Service, SPARQL Graph Service, and Embedding Service. The Main Application component serves as the entry point of the application. FastAPI was used as a wrapper to optimize performance and enhance the agent's robustness and flexibility. The user input is then forwarded to the Answering Service, which uses the Extraction Service to extract the entity and relation from the user prompt. After identifying the entity and relation, the SPARQL Graph Service and Embedding Service are used to retrieve and calculate answers for the user's question. The final answer is then returned to the Main Application component. In the following subsection, each individual step is explained in more detail.

2.1 Factual Questions

Our agent addresses factual questions through the following steps:
Identification of Entities and Movies

- **Movie Identification:** First, we identify movies within the sentence if a match is detected. This process consists of two steps:
  - We begin with a rule-based approach using the NLTK library[1] to match movie names. If an exact match is not found, we use a fuzzy

[1] `https://www.nltk.org/`

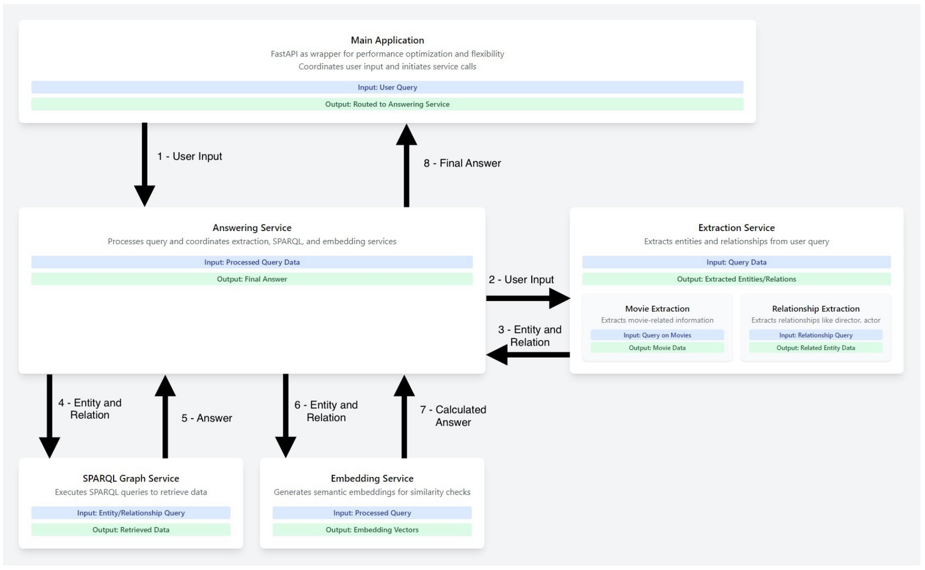
Main Application
FastAPI as wrapper for performance optimization and flexibility
Coordinates user input and initiates service calls
Input: User Query
Output: Routed to Answering Service
1 - User Input
8 - Final Answer
Answering Service
Processes query and coordinates extraction, SPARQL, and embedding services
Input: Processed Query Data
Output: Final Answer
2 - User Input
3 - Entity and Relation
Extraction Service
Extracts entities and relationships from user query
Input: Query Data
Output: Extracted Entities/Relations
Movie Extraction
Extracts movie-related information
Input: Query on Movies
Output: Movie Data
Relationship Extraction
Extracts relationships like director, actor
Input: Relationship Query
Output: Related Entity Data
4 - Entity and Relation
5 - Answer
6 - Entity and Relation
7 - Calculated Answer
SPARQL Graph Service
Executes SPARQL queries to retrieve data
Input: Entity/Relationship Query
Output: Retrieved Data
Embedding Service
Generates semantic embeddings for similarity checks
Input: Processed Query
Output: Embedding Vectors

Figure 1: Diagram showing the input and output of each service.

search with the FuzzyWuzzy library[2], leveraging Levenshtein distance for improved performance, and apply a similarity threshold of 90. This allows for identifying movie names even when they are not exact matches to known labels. One challenge involves certain movie names that contain terms shared with other movies (e.g., "Star Wars: Return of the Jedi" contains the term "Return," which could also refer to another movie). To address this, we implement an algorithm that extracts the longest matching sequence, prioritizing specificity in identification.

- **Relation Identification:** Once a movie is identified, we proceed to identify relations. Initially, we apply a rule-based NLTK approach to match extracted labels for all relations associated with the movie. If no direct match is found, we use SentenceTransformers[3] to embed all relations from the knowledge graph into a vector space. During this process, stop words are filtered out, and the movie name is removed from the context. We identify a relation match when the cosine similarity exceeds a threshold of 0.5. This threshold is intentionally set higher, as this mechanism serves as a fallback and is intended to produce only highly relevant responses.

[2]`https://pypi.org/project/fuzzywuzzy/`
[3]`https://sbert.net/`

2.2 Embedding Questions

If the answer cannot be retrieved from the knowledge graph, an attempt is made to calculate the answer using embeddings. The implementation is based on examples provided in the given tutorials. First, the entity vector and relation vector are identified. These vectors are then combined according to the TransE scoring function. Subsequently, the distance is computed, and the most plausible entity is selected. The answer is then embedded into a sentence and returned to the user. If an answer to the user's question cannot be found, a request is made for the user to rephrase the question or to try a different one.

It is assumed that the knowledge graph contains only valid answers; therefore, answers retrieved from the graph are not cross-checked with embeddings. Embeddings are used solely as a backup, as they are generally less accurate.

3 Adopted Methods

- **NLTK**
  We used the NLTK library[4] to implement a rule-based approach for identifying labels, specifically for movies and relations. Additionally, NLTK is used to remove stop words from input sentences, helping to improve the accuracy of relation extraction by reducing noise.

- **SentenceTransformer Library**
  The SentenceTransformer library[5] is used to embed relations into a vector space, allowing us to compare the input sentence with known relations. We specifically utilize the 'all-mpnet-base-v2' model[6] from Hugging Face, which is well-suited for semantic similarity tasks. The input sentence is preprocessed by removing stop words to ensure the embeddings focus on key terms, enhancing the relevance of similarity comparisons.

- **FuzzyWuzzy and Levenshtein Distance**
  We leverage the FuzzyWuzzy library[7] for fuzzy string matching, which allows us to match movie names even when they are not exact matches. FuzzyWuzzy relies on the Levenshtein distance algorithm[8] to calculate the similarity between strings. For improved accuracy, we set a similarity threshold of 90%, ensuring that only closely matching terms are identified as movie names, even if they contain minor variations.

[4] `https://www.nltk.org/`
[5] `https://sbert.net/`
[6] `https://huggingface.co/sentence-transformers/all-mpnet-base-v2`
[7] `https://pypi.org/project/fuzzywuzzy/`
[8] `https://en.wikipedia.org/wiki/Levenshtein_distance`

4 Examples

4.1 Factual Questions

Given a question such as, "*Who is the director of Good Will Hunting?*" our agent follows a structured process to retrieve the answer:

First, the agent identifies the movie entity—in this case, *Good Will Hunting*—using the NLTK library for movie extraction. Once the movie entity is detected, it is masked in the question to focus on identifying relationships. In this example, the relationship *director* is extracted as the key relation.

Next, these extracted labels are mapped to their corresponding entity and predicate in the knowledge graph. Once the mapping is complete, the agent constructs and executes a query in the knowledge graph, returning the result as the answer.

4.2 Embedded Questions

If the question cannot be answered using the graph, the answer is calculated using embeddings. Since the graph is assumed to contain only correct information, embeddings are used solely as a backup. Therefore, no cross-checking is performed, and the answer with the highest score is selected and returned.

5 Additional Features

To enhance the timeliness of our agent, we preload all necessary models and libraries upon startup on the server. For development purposes, we implemented a lazy-loading approach, allowing for a faster application startup during testing and debugging. Additionally, we pre-exported all movie labels and relationships to avoid redundant processing, loading them into memory as predefined data for quicker access.

The *humanness* of our agent is achieved through a response rendering stage, where we defined five distinct response templates. These templates are dynamically filled with the answer to produce responses that feel more natural and conversational. Additionally, we add manual checks for *Welcome* and *Thanks* messages, to which we respond without performing query.

One challenge we addressed involved the varying wording of relationships and the use of synonyms. For example, a user might ask, "When was a movie released?" whereas the corresponding relationship in the knowledge graph is labeled as *publication date*. To bridge this gap, we initially used embeddings to match synonyms more accurately. However, as this approach did not yield sufficient precision, we implemented a manual mapping specifically for *publication date* to correctly align it with *release date* in the user's query.

Another challenge we encountered involved hyphens and their variants. To improve search accuracy, whenever the incoming query contains a hyphen, we automatically replace it with different variants—such as em and en dashes—to improve matching accuracy. This solution has significantly reduced potential user errors related to hyphenation, making the user experience more resilient to variations in the dataset.

We also experimented with BERT [1] for Named Entity Recognition (NER) to enhance entity extraction accuracy. Although this approach required additional training on our dataset, it still relied on matching with fixed labels in the knowledge graph—primarily through fuzzy search. Given these dependencies, we ultimately opted for a pure fuzzy search approach combined with rule-based matching using NLTK, which provided more consistent results with less complexity.

6 Conclusions

For the next steps, we plan to explore recommendation algorithms to enhance our chatbot's capabilities. Additionally, we aim to improve the extraction pipeline to handle multiple movies within a single query more effectively. As our chatbot will support both recommendation and knowledge-based questions, we need a mechanism to distinguish between the two types. Rather than relying solely on the number of movies extracted, we intend to deploy a small neural network that can accurately classify the question type and determine the appropriate response approach.

References

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.