Post-processing and validation of LLM outputs using knowledge bases

Angela Atem (2782550) Akili van Heyningen (2627924) Vajiheh Moshtagh (2801823) Ivana Pleše (2862350)

***Abstract*—Large Language Models (LLMs) are computational models designed for natural language processing tasks that are used to generate and understand text. They learn patterns and structures from large datasets and can produce impressive results. However, LLMs also inherit biases from their training data, making post-processing and validation crucial when working with them. The goal of this project was to address this problem by designing a program to post-process the outputs generated by large language models.**

I. Introduction

For the task of post-processing, extracting answers and validating their correctness we've used Meta's LLaMA 2, and Wikidata as our KB of choice.

II. Problem

The first step involves extracting entities (such as people, places, or things) from the response generated by the LLM. This is achieved using the SpaCy NLP library, which identifies named entities in the text. Once the entities are identified, each entity is linked to its corresponding entry in the Wikidata knowledge base. This is done by querying the Wikidata API to search for candidate entities and ranking them based on similarity to the extracted entity. The ranking considers the entity's label, aliases, and description, as well as the context in which the entity appears.

The second step involves verifying the accuracy of the linked entities. After ranking the candidate entities, the best match is selected, and the corresponding Wikipedia link is fetched. This validation ensures that the extracted entities are correctly identified. It is a crucial step in minimizing the risk of associating incorrect or ambiguous entities with the generated response. Recall is very important in this step.

The third step focuses on extracting answers from the response generated by the LLM. This step uses the RoBERTa question-answering pipeline to extract direct answers from the cleaned LLM response.

In the fourth step, we perform fact-checking using RoBERTa. We also measure entity and context similarities and gives a confidence score for the judgment.

III. Methods

Models and tools used:

- LLama (Llama-2-7b) Model
- SpaCy (en_core_web_sm)
- Wikidata API
- Cosine Similarity (TF-IDF + Sklearn)
- SequenceMatcher (from Python's difflib library)
- RoBERTa
- TensorFlow

A. Details of the implementation

Named Entity Recognition We use SpaCy's built-in NER tool. SpaCy's NER model uses several techniques to identify entities in text: word vectors are used to represent tokens, POS tagging for named entities and dependency parsing to understand relationships between words.

We have explored some pre-processing techniques to improve the performance of the NER model:

- **stop-word removal:** We created our own extensive list of stop-words and removed them before sending the text to SpaCy.
- **Hamming distance:** We implemented our own Hamming distance function to compare string similarity.
- **removing leading articles and recognizing capital words:** We developed a custom function to identify and remove leading articles from the text.
- **abbreviations patterns:** We implemented custom regular expressions to identify abbreviations and capitalized words.

Despite these pre-processing efforts, we found that sending the raw text directly to SpaCy without any additional pre-processing achieved the best performance in terms of NER accuracy.

Named Entity Linking

For each entity detected in the input, the system queries Wikidata using Wikidata API to find possible matching entries. Candidates are retrieved on the basis of their labels. For the candidate list, another call is made to fetch entity aliases based on the candidate ID. Both calls retrieve additional information about the candidates.

Candidates are ranked based on their similarity to the context of the text, such as a sentence or query. The ranking takes into account two main factors. The first is the title match score, which measures the similarity between the candidate's label and the entity's label. If the labels are identical or an alias matches, the score is high. If not, a string similarity score is calculated using difflib.SequenceMatcher. The second factor is context similarity, which compares the candidate's description from Wikidata with the surrounding text context from the entity mention in the query. This is done using

cosine similarity of their text embeddings, which are generated using the TfidfVectorizer from sklearn. The final ranking is determined by a weighted sum of the title match score and context similarity score, with a higher overall score indicating a better match. The candidate with the highest ranking will be matched to a Wikipedia link and if there is no such link, it will be matched to a highest-ranking sitelink.

After several tests, we fine-tuned the weights assigned to a scoring function and managed to find a balanced set of weights that produced more accurate rankings for the candidates. This improved the entity linking process.

We attempted to speed up the retrieval of entity candidates from an external API by parallelizing the question processing and candidate fetching from Wikidata.

Answer Extraction

We focused on extracting answers from the output generated by the LLM, keeping track of the type of question imposed in the original query. For yes/no questions, we used a combination of keyword spotting and syntactic analysis to determine whether the response affirms or denies the query. We scanned the response for explicit strings such as "yes," "surely," or "no," "not at all," and checked for negation words in sentences using the SpaCy library.

Questions with specific entity answers were done in two steps:

1. **Named Entity Recognition (NER):** We deploy SpaCy's NER capabilities to identify and extract potential entities from the LLM's response. This step is crucial as it forms the basis for further validation against external knowledge bases.
2. **Contextual Relevance Check:** After identifying potential entities, the system evaluates the context in which each entity appears to ensure its relevance to the question. This is achieved by analyzing the surrounding text and comparing it to the question's focus, thus filtering out irrelevant or incorrectly recognized entities.

The accuracy of this module is critical, as it directly affects the reliability of the fact-checking process that follows. By extracting and validating the entities or answers, we ensure that the subsequent verification step is based on correct and relevant data. This methodology not only enhances the integrity of the answers provided by the LLM but also minimizes the propagation of errors in downstream applications.

Fact Checking

To validate answers, we check if the extracted answer matches any of the known entities in the entities dictionary. The dictionary contains entities and their corresponding values, and if the extracted answer is found in it, it is considered valid. If the answer matches, the validation result is marked as valid and the confidence score is carried over. Both extracting the answers and measuring the confidence score are done by RoBERTa.

In the fact-checking segment of the solution, our initial idea was to use Stanford OpenIE extracted triples, which we planned on creating in the answer extraction phase of the project. The goal was to analyze the Wikidata page of the object entity and its properties to extract the most relevant property for the subject entity. To do that, we tried embedding the relation and all the properties linked to the object entity to later make a comparison using a cosine similarity. The system would validate the extracted answer by cross-referencing it with the linked entities in Wikidata. The best result would then be checked to determine whether it contains the subject entity. That would be our basis for verifying the truth.

Even though we wrote code that would perform most of the tasks needed to implement this part of the solution, we weren't able to extract triples needed for this implementation to work. We were counting on having a function that would translate open-ended questions to a form more suitable for open relation extraction and classify the type of questions used in the queries. In the end, we decided to make a simpler implementation.

B. Parallelized Processing of Questions

To have the code run efficiently and be more scalable, we parallelized the question processing pipeline. The questions are distributed across multiple processes, enabling concurrent computation. Each process executes the following steps:

1) Generate a response for the query using LLaMa-2-7b model
2) Extract entities and link them to Wikidata entries
3) Perform answer extraction using RoBERTa and validate the correctness of the response

The multiprocessing architecture minimizes latency. This approach was expected to run faster; however, the empirical evaluation showed variability in runtime. For example, processing 7 questions typically took around 10 minutes, while processing 10 questions ranged from 8 to 20 minutes. Interestingly, there were cases where 8/10 questions were completed within 8 minutes, demonstrating inconsistency in run-time performance.