Rachel Benincasa, Cole Hersh, Bronson Housmans, Shayden Lowry
CSC 483
Final Project Proposal and Implementation

1. General Project Idea

The goal of this project is to build an information retrieval system that can answer Jeopardy-style trivia questions by selecting the correct Wikipedia article title as the answer. For this project, we are given a collection of around 280,000 full Wikipedia articles, and our program must return the title of the article that best matches the answer to the question given.

The final outcome will be a search-based solution with multiple steps. The approach we want to take is to have multiple steps of preprocessing and indexing through the articles to get the most relevant information from each article, which will eventually lead to having an LLM look at the top X articles (X to be decided), and from there select the correct answer. Our solution will be evaluated on accuracy, or the percentage of questions where the title matches the correct answer. We are aiming to get at least 40/100 or 40% accuracy, but we will aim for higher accuracy if possible.

2. The Problem

For our inputs, we are given:

- A trivia question (ex. “The city known as the Big Apple”)
- 280,715 Wikipedia articles

Our output will be:

- The title of the most relevant Wikipedia article (ex. “New York City”)

Our task is to match natural language questions to article titles. The main challenge comes from the fact that the articles are not standardized and may be weirdly formatted. Some examples of this could be having indirect references, metaphors, or other similar things that may make the processing section incorrectly think that an article is relevant when it isn’t, and vice versa.

For this project, we are given a list of 100 trivia questions to use as our inputs, and we must use those to see the accuracy of our code. However, the code should work for any trivia question.

3. Dataset Description

The dataset we are given is a Wikipedia article collection that contains 280,715 articles. There are a few dozen files containing all of the text for each of the articles, and each of these files is in a folder called wiki-subset-files. Each article is written in plain text with specific markups that indicate titles, sections, and other decisions.

Each article has a title, which is shown in double brackets (ex. [[Title]]). This makes it straightforward to extract the exact title name from the article, which will be needed as it is the target output for the problem we want to solve. The body of the article is organized into sections using equal signs (ex. === Main Section ===). There are often multiple sections for each article, but some also have very little text in them. This structuring allows for easy access to see specific things about the topic / article, and also allows for potentially weighing earlier sections more, as they are potentially more important than later sections. There are many articles with the opening line #REDIRECT to point to another wiki page. For the sake of simplicity, we decided to ignore these documents for this project.

Aside from this, there are other annotations that are less useful for our processing. A few are CATEGORIES:, citation tags such as “[tpl]cite web[...]”, and template language. We assume these, for the most part, can be ignored as they usually do not hold relevance for the problem we are trying to solve, and instead, we want to focus on the body of the text itself.

4. Baseline

For our project, it was important that we had a starting point to be able to compare our future code with to make sure we see improvement. The first baseline we tried is simply random guessing, or picking a random article for each input given. With 280,715 articles and only one correct answer, the chance of getting just one right is 1/280,715, or around 0.00036%. Paired with the fact that we have 100 questions to test, this model results in a basically 0 chance of getting one right.

The next baseline we implemented was using only tf-idf weights to score all of the documents in our dataset for each of the 100 queries. The results from this scoring method gave 4 correct answers when the document returned was the one with the highest tf-idf weight.

Going into the implementation of our jeopardy game, our baseline was 4%, and we are aiming for an improvement, and at least a 40% success rate for the 100 answers we are searching for.

5. Proposal and Results

Our proposed solution has 2 main steps. First is a traditional information retrieval approach using TF-IDF (and a similar scoring method called BM25F), followed by using an LLM to get the correct answer.

To start, we cleaned up the text. We removed punctuation, special characters, URLs, and a good amount of stop words. Next, it uses lemmatization via the SpaCy library to reduce words to their base forms (ex. Running becomes Run), and removes any unnecessary words that don't add much meaning to the article. We also loaded the articles from the directory containing the files (wiki-subset-text) and extracted the article title and the cleaned-up text, and stored them for future processing in JSON files.

From there, we implemented a TF-IDF (and BM25F) index for each article. We compute both the term frequency and the inverse document frequency. We also use TF-IDF on the questions being asked, and the same process is done to the questions (cleaning up the text, lemmatizing, computing TF and IDF, etc). We then use these values to score articles based on how closely their content matches the question.

In our process of creating this implementation, we wanted to retrieve the top X values from the TF-IDF process, and spent a lot of time seeing what value of X (10, 50, 100, 500, etc) would give us the best result. We wanted to make sure that the top X value did in fact contain the correct article, but we didn't want X to be too big that it becomes hard to run or doesn't give any extra benefit. In our **query** method, we included a parameter for this value to allow both testing of multiple values and also the ability to set it to whatever the user would like.

We found that our own implementation of tf-idf was not as robust as we would have liked, with the results coming in at 4% correct documents for the top result, 30% of correct documents being present in a set of the top 20 tf-idf scores, and 36% of correct document being present in a set of the top 50 tf-idf scores. Further tests were run with set sizes greater than 50, but this proved to have no significant improvements and vastly increased the runtime. In response to these results, the group decided to implement a different search engine. After a failed attempt to install the PyLucene library, research led us to try the Whoosh library, which is completely written in Python.

The Whoosh library was able to create indexes on all of the JSON files stored with the preprocessed data to greatly improve the search time and results for the 100 queries. We found that when taking the top 20 scores from a search with whoosh, 53% of correct documents were present in the set for the question with no textual processing, 56% of correct documents were present in the set for the question with the textual processing performed on the wiki pages, and 62% of correct documents were present in the set for the question with the textual processing. This vast improvement over the previous engine led us to take the results provided by this library and return arrays for each query with the 20 documents with the largest scores.

Once we have a set of the top 20 articles from the better-performing search engine, we take those articles and feed them into an LLM, specifically LLAMA. We gave it specific instructions on how to pick the correct answer ("The format of the input is a dictionary with Category, Question, and DocNames as the keys. The DocNames key has the associated value of document names in an array. You are only allowed to return one of the 20 document names present. "). From there, the LLM is able to take in the set of inputs and give one title as an output.

Our initial result when using this method was around 36% for the Hermes-3-Llama-3.2-3B-GGUF model, with variation of around 2-3% for a given run. One way that the group decided to increase the accuracy of the model was to provide additional system prompts for the language model. These were added for Categories that were found to be challenging for the language model to understand, primarily “TIN” Men, UCLA Celebrity Alumni, and Golden Globe Winners. This increased the accuracy of the documents to 38% on average, with a peak result of 43%. However, we wanted to reach 40% since the results were so close, and we decided to use the model Llama-3.1-8B, since it has been trained on a larger number of parameters (8 billion as opposed to 3 billion for the other model). With some changes to the prompt and by setting the temperature to 0 to reduce the inconsistency in our resulting runs, we were able to get 48% accuracy using this model, with a slightly longer runtime.