Tulane Academic Advisor With RAG

Gavin Galusha
Tulane University / CA
ggalusha@tulane.edu

Gabe Epstein
Tulane University / CO
gepstein1@tulane.edu

Abstract

We created a Retrieval Augmented Generation system with the goal of answering Tulane specific academic questions. By web-scraping Tulane's website, we were able to create a generative question answering system that was knowledgable on two main sectors: Tulane Programs, and Tulane Courses. We paired this retrieval system with a state of the art LLM (gpt-3.5-turbo) using it's chat completion feature to generate high quality responses to Tulane specific queries. After experimenting with a variety of retrieval methods, we landed on a chunking technique for the documents, followed by pre-processing, and dynamic top k retrieval, which proved to be a reliable way to obtain the relevant context for the LLM. To make our system easily usable, we implemented a web interface using flask, which allows users to toggle which data they want to interact with, and ask questions in an intuitive manner.

1 Introduction

Contacting academic advisors can be an arduous process. Advisors are often bombarded with easily answerable questions, they can be hard to get ahold of, or perhaps simply unqualified for their position. Our goal was to create a system that would streamline information from readily available public information online, directly to a student.

The emphasis on this project was to build a robust retrieval system, so that in the future as more and more data is available, the system improves relationally.

2 Approach

Web Scraping

Our first step in the overall process was to collect the relevant data. We used Beautiful Soup to parse the Tulane courses page, where we were able to extract relevant text data. After obtaining a dictionary of each program name as the keys, and the associated links as the values, we parsed the two main pages for each program, the requirements and home tabs found under each program. From here, we concatenated all the relevant text, and saved it to a directory where each program was divided into text files based on the topic. The result was a directory of over 400 files, with a hearty amount of text dedicated to each program.

A similar procedure was done for the Tulane courses page, only we had to employ further text manipulation to accurately extract each specific class offered. We used regex matching to create a new entry every time four capital letters were seen, indicative of class descriptions like CMPS and ACCN. We did further filtering out for courses that had no descriptions, or special courses like "Independent Study" which would offer no relevance in our retrieval system. The output of this were over a thousand very small class files, each with the course code, and whatever information about the class was given.

This web scraping process used no hard encoded values besides the links to the websites, so the web scraper can easily be run and obtain any new courses, programs, or just general information published on the Tulane website.

Data Preparation

To turn our text file data into an easily retrievable format, we first needed to create embeddings for the documents. During data preprocessing, we used the Spacy library to "clean" the text in each document. This included normalizing the text to all lowercase, removing punctuation and stop words, and applying Lemmatization. In this process, we also used Named Entity Recognition to determine if each entity in a document was the name of a program or class, in which case the entire phrase/name would be added to the processed text output, rather than a stripped version.

This pre-processing is an essential step to generate richer embeddings for the documents, aiding in effective retrieval.

Chunking

To further improve retrieval, we employed a chunking technique to split up each document into smaller parts. We found success with relatively large chunks, around 5000 characters, with 100 character overlaps. This improved the number of documents retrieved from around 9-13 to 20-30 for the programs data. No chunking was needed for the course data as documents are already sufficiently small.

Vectorization

We experimented with dense embeddings created by models like BERT and the Openai model "text-embeddings-3-small", but concluded that they provided no significant advantage over simple TF-ITF vectorization.

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Retrieval

The actual retrieval method used was a combination of cosine similarity and keyword priority. the retrieval takes an input query (a sentence) and vectorizes it into the same space as the document embeddings using TF-ITF again. From here we calculate a score for each document based on the cosine similarity of the vectors.

$$similarity(A,B) = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sum_{i=1}^{n} B_i^2}.$$

We add this to the keyword score, which is calculated by taking the set of all word in the query, and searching for those words in the title of each document. This final score is then added to a list called matches with the associated document.

The next step in retrieval is to return the top_k to form the context for the LLM. In our case, the LLM we are using has a context window of around 16,000 tokens, so in order to always fill that context window, we employed a dynamic top_k approach.

Dynamic Top_k

To optimize the amount of documents retrieved with each query, we used tiktoken to count the number of tokens appended to the context window. This way, we can start at the top of the matches list, getting the documents with the highest combined scores, and iteratively retrieve more documents until the maximum token limit is reached. This step proved essential, as it is effective on databases of documents with greatly varying lengths, and no top_k variable is needed to be hardcoded.

Experiments

Our experiments led us to many conclusions that would aid our project. The first was an attempt to combine both the courses and programs data in a top_k format. This quickly turn problematic, as the top_k documents may easily go far over or far under the context window, as document length varied too drastically. After implementing the dynamic top_k strategy to deal with this issue, we still found confounding results. Providing a slew of courses data with programs data seemed to be a confounding context for the LLM, as the response output accuracy took a sharp fall. Questions the system had easily answered like "What are the major requirements for x major" were answered, but with a jumble of information not consistent with the actual website specifications. This led to the idea of separating the question answering into two modes, class expert, and programs expert. These two modes isolate the data that can be retrieved in the process, proving to faster and more accurate.

Another critical in the experimenting process was to save the pre-processed data to a file for easy use. The pre-processing function takes up to a minute

Retrieval

The actual retrieval method used was a combination of cosine similarity and keyword priority. the retrieval takes an input query (a sentence) and vectorizes it into the same space as the document embeddings using TF-ITF again. From here we calculate a score for each document based on the cosine similarity of the vectors.

$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sum_{i=1}^{n} B_i^2}.$$

We add this to the keyword score, which is calculated by taking the set of all word in the query, and searching for those words in the title of each document. This final score is then added to a list called matches with the associated document.

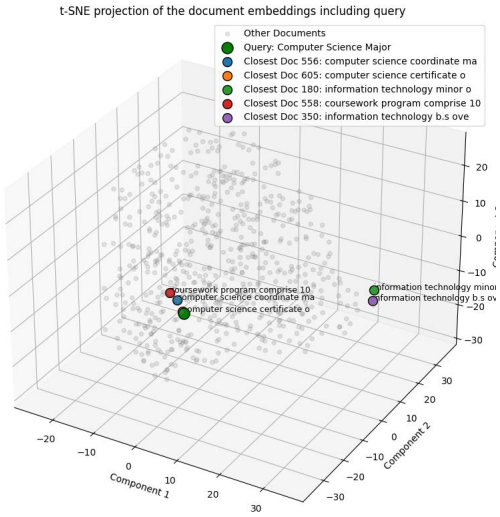
The next step in retrieval is to return the top_k to form the context for the LLM. In our case, the LLM we are using has a context window of around 16,000 tokens, so in order to always fill that context window, we employed a dynamic top_k approach.

Dynamic Top_k

To optimize the amount of documents retrieved with each query, we used tiktoken to count the number of tokens appended to the context window. This way, we can start at the top of the matches list, getting the documents with the highest combined scores, and iteratively retrieve more documents until the maximum token limit is reached. This step proved essential, as it is effective on databases of documents with greatly varying lengths, and no top_k variable is needed to be hardcoded.

Results

The resulting functionality of the retrieve method is accurate at retrieving the relevant chunks of data. In this example we reduce dimensionality to 3d space, and as an example show the 5 most similar chunked documents to the query: "Computer Science Major"



The funct

Retrieval

The actual retrieval method used was a combination of cosine similarity and keyword priority. the retrieval takes an input query (a sentence) and vectorizes it into the same space as the document embeddings using TF-ITF again. From here we calculate a score for each document based on the cosine similarity of the vectors.

$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sum_{i=1}^{n} B_i^2}.$$

We add this to the keyword score, which is calculated by taking the set of all word in the query, and searching for those words in the title of each document. This final score is then added to a list called matches with the associated document.

The next step in retrieval is to return the top_k to form the context for the LLM. In our case, the LLM we are using has a context window of around 16,000 tokens, so in order to always fill that context window, we employed a dynamic top_k approach.

Dynamic Top_k

To optimize the amount of documents retrieved with each query, we used tiktoken to count the number of tokens appended to the context window. This way, we can start at the top of the matches list, getting the documents with the highest combined scores, and iteratively retrieve more documents until the maximum token limit is reached. This step proved essential, as it is effective on databases of documents with greatly varying lengths, and no top_k variable is needed to be hardcoded.