Project Title: "Building a Smart Search System for Analytics Vidhya Courses Using Embeddings and LLMs"

Submitted by

-Bheemisetty Pavan Sai

12112111

Pavansai.bheemisetty@gmail.com

Date : 23-10-2024

Deployed project link :

https://huggingface.co/spaces/PavanSai25/AnalyticsVidhya_searchTool

Github link (Contains All files):

https://github.com/PavansaiBheemisetty/AnalyticsVidhya_SearchTool

Table of contents:

3. Introduction

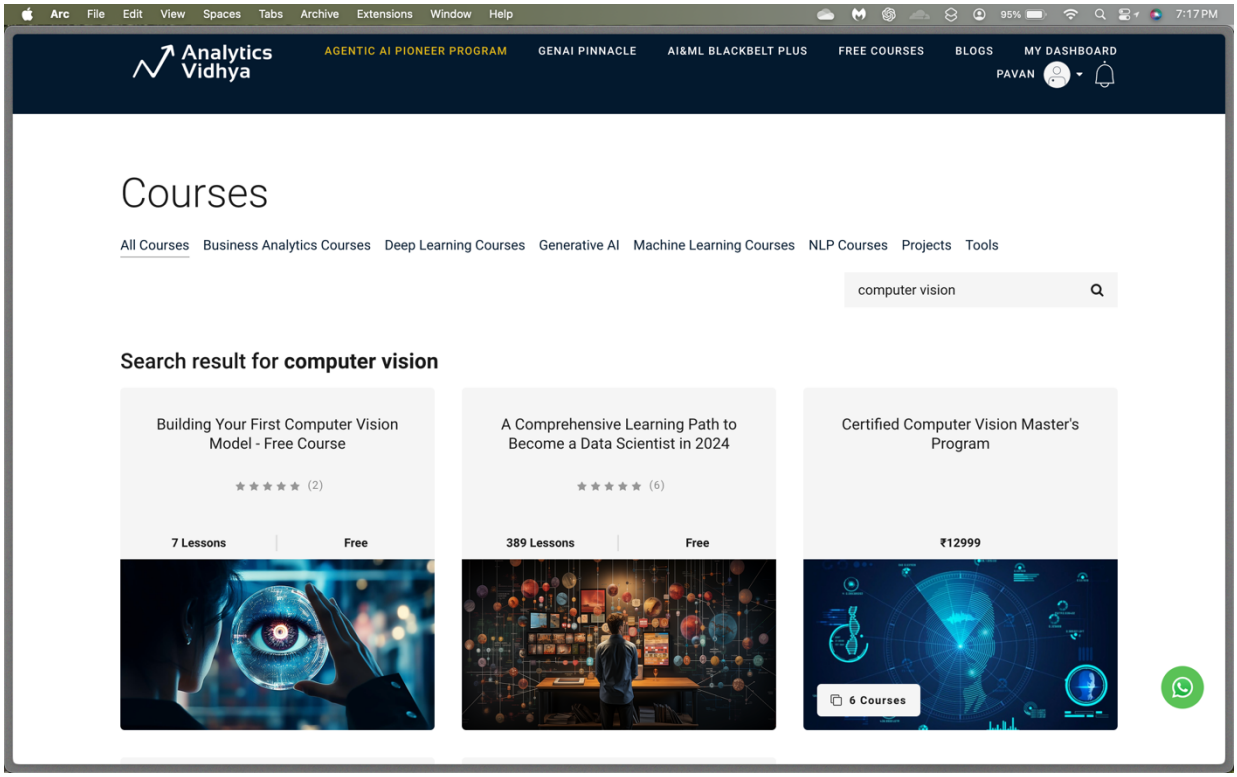
In today's digital learning landscape, navigating the extensive range of available resources can be quite daunting. As online educational platforms expand, there is an increasing demand for more sophisticated and intuitive search functionalities that surpass basic keyword matching. This initiative seeks to fulfill that demand by creating an intelligent search system for Analytics Vidhya's free courses, utilizing cutting-edge methods in machine learning and natural language processing.

The objective is to offer users a streamlined and intelligent approach to discovering courses that align with their learning preferences, even when their search terms do not precisely correspond to the course titles or descriptions. By employing embeddings and large language models (LLMs), this search system effectively captures the deeper semantic meanings of both user queries and course content, resulting in more precise and contextually appropriate search outcomes.

This report outlines the development process of the smart search system, covering aspects from data collection and preprocessing to model selection and deployment. It aims to provide a thorough understanding of the methodologies employed to connect user intent with search results, enhancing the search experience for learners.

4. Problem Definition

In the current digital environment, where online education is gaining traction, users frequently find it challenging to locate courses that align with their individual needs and preferences. The main objective of this initiative is to develop an intelligent search system that enables users to easily find free courses available on the Analytics Vidhya platform. Given the extensive selection of courses, it is crucial to simplify the search process, ensuring it is both user-friendly and effective.



Keyword Search vs. Natural Language Search:

- **Keyword Search**: Searching for exact words from the query. For example, searching for "Machine Learning" will return courses containing the phrase "Machine Learning."
- **Natural Language Search**: Interpreting a user's query in a more human-like way. For instance, if someone searches for "How do I get started with data science?", the search should find beginner-level data science courses

User Challenges

Numerous users face obstacles when attempting to search for courses due to ambiguous or complicated queries. For example, a user interested in machine learning may struggle to express their requirements in a search query, potentially overlooking valuable resources that could enrich their learning journey. Additionally, conventional keyword-based search systems often yield irrelevant results, resulting in user frustration and diminished engagement with the platform.

Essential Requirements

To tackle these issues, our intelligent search system must fulfill several essential requirements:

Interpreting Diverse User Queries:

The system should proficiently interpret and analyze a wide range of user inputs, including natural language queries, to grasp the user's intent. This involves accommodating synonyms, variations in wording, and even incomplete queries.

Delivering Relevant Search Results:

The search feature must provide course results that closely match the user's query. This requires a comprehensive ranking system that evaluates various factors, such as course relevance, popularity, and user ratings.

Maintaining Scalability:

As the number of courses on the Analytics Vidhya platform continues to expand, the system must be scalable to manage the growing data without sacrificing performance. This includes efficiently indexing and retrieving a larger volume of courses while ensuring rapid response times for users.

5. Data Collection

The initial phase of this project involved gathering course data from the Analytics Vidhya website. The goal was to compile comprehensive information that would feed into our smart search system. To accomplish this, I utilized two popular web scraping libraries: Beautiful Soup (bs4) and Selenium. However, I encountered significant challenges along the way.

Attempt with Beautiful Soup (bs4)

I started by using Beautiful Soup, a powerful Python library for parsing HTML and XML documents. My plan was to scrape course details directly from the webpage’s source. However, the results were disappointing. The structure of the website was more complex than anticipated, and many of the course details were dynamically loaded via JavaScript after the initial page load. Consequently, Beautiful Soup could not capture this dynamically rendered content, resulting in incomplete or missing data.

Second Attempt with Selenium

Realizing the limitations of Beautiful Soup, I shifted to Selenium, which allows for browser automation and can interact with JavaScript-rendered content. While I hoped this would provide a more robust solution, I still faced challenges. The

website employed anti-scraping measures, including rate limiting and content delivery via AJAX calls, which made it difficult to access the desired data efficiently. Despite multiple attempts to adjust my scraping strategy, including modifying the wait times and using different browser configurations, I was unable to retrieve the course data successfully.

Manual Data Entry

After these setbacks, I decided to take a different approach. Instead of relying on automated scraping, I manually entered the course information into an Excel sheet. This method allowed me to ensure the accuracy and completeness of the data. I created several columns to capture key features, including:

- **Course Title**: The name of the course, which helps users identify relevant content.
- **Level**: The difficulty level of the course (e.g., Beginner, Intermediate, Advanced), allowing users to select courses that match their skill set.
- **Time**: The estimated duration to complete the course, which helps users gauge their commitment.
- **Category**: The subject area of the course, aiding in organizing content and improving search relevance.
- **Number of Lessons**: The total count of lessons included in the course, providing an indication of the course's depth.
- **Description**: A brief overview of what the course entails, helping users make informed choices.
- **Curriculum**: Detailed information about the topics covered in the course, allowing for a deeper understanding of course content.

In total, I manually compiled **59 entries** into the Excel sheet, ensuring that all essential information was accurately represented. This data will serve as the foundation for developing the search functionalities of our system.

6. Data Preprocessing

After collecting the initial dataset, I proceeded to the data preprocessing stage to prepare the information for embedding and search functionalities. Given the manual data entry approach, certain inconsistencies and errors needed to be addressed before further processing.

Handling Mistakenly Entered Paid Courses

While entering the data, I accidentally included two rows corresponding to paid courses, which were not relevant to the project's goal of building a smart search system for free courses. To flag these rows for removal during data cleaning, I intentionally left some columns empty for these two entries. This allowed for easy identification and ensured that they could be excluded from the final dataset without affecting other data points.

Data Quality and Cleaning

The dataset consisted of **59 entries**, with no duplicates present. During the cleaning process, I focused on:

- **Dropping the rows of paid courses**: Using the empty columns as indicators, I removed the two unwanted rows to maintain the dataset's relevance.
- **Handling Missing Values**: Since the columns left empty were meant for dropping specific rows, there were no unintentional missing values in the rest of the dataset.

Simplified Approach to Text Processing

Given that the data was entered manually, it was already fairly clean and consistent, so I opted for a simplified preprocessing approach:

- **No Tokenization**: I decided not to perform tokenization or other advanced text preprocessing steps, as the existing data was well-structured and minimal preprocessing was sufficient for the embedding model.
- **Normalization Not Required**: As the manually entered text did not contain special characters, symbols, or inconsistencies that would typically require normalization, I skipped this step.

This straightforward approach to data preprocessing ensured that the dataset was ready for embedding while minimizing unnecessary alterations that could affect the quality of the search results.

7. Embedding Model Selection

When selecting an embedding model, my goal was to find one that could create meaningful representations of text data to enhance the search experience. After evaluating several alternatives, I opted for the all-MiniLM-L6-v2 model, a pre-trained option from the Sentence Transformers library. Here are the reasons this model was the most suitable choice for the project.

Reasons for Choosing all-MiniLM-L6-v2

The main factor in choosing the all-MiniLM-L6-v2 model was its capability to generate compact yet highly informative embeddings. This model excels at efficiently capturing the semantic essence of text while keeping the embedding size small, which is essential for scaling the search system without sacrificing performance. Given that the dataset currently comprises 59 courses but has the potential for significant growth, it was vital to select a model that strikes a balance between accuracy and computational efficiency.

Additionally, all-MiniLM-L6-v2 is tailored for sentence-level tasks such as semantic search, making it an ideal match for the project's needs. It effectively transforms course titles, descriptions, and other attributes into embeddings that convey the underlying meaning, facilitating the retrieval of relevant results even for diverse or unconventional user queries.

Considerations Against BERT or RoBERTa

At first, I explored other well-known models like BERT and RoBERTa, which are also pre-trained language models recognized for their performance in NLP tasks. However, they posed certain challenges:

Size and Efficiency: Both BERT and RoBERTa produce larger embeddings and contain more parameters than all-MiniLM-L6-v2. For this project, I required a model that would not demand excessive computational resources, particularly since the intention was to deploy the system on a lightweight hosting platform like Hugging Face Spaces. Utilizing larger models would have resulted in increased latency during searches and higher hosting expenses, which was not desirable.

Use Case Specificity: While BERT and RoBERTa are effective for a range of NLP tasks, they are not specifically optimized for the requirements at hand.

8. Indexing Using FAISS

After generating embeddings for the courses, the next challenge was figuring out how to search through them efficiently. I needed a way to quickly find similar courses based on a user's query, and that's where FAISS (Facebook AI Similarity Search) came in. It's a library specifically designed for fast, approximate nearest-neighbor searches in high-dimensional spaces—basically, a perfect match for what I needed.

Why I Chose FAISS

I went with FAISS for a few simple reasons. First, it's **optimized for speed**, which is super important when you're dealing with embeddings, even if it's just 59 courses for now. I didn't want the search to slow down as the data grows, and FAISS is known for being able to handle **large-scale searches efficiently**.

Another big factor was **memory efficiency**. FAISS supports different indexing techniques, including ones that use less memory or compress the data, which means it can scale up without using tons of resources. This was important because, as I plan to add more courses, I want the system to keep up without needing major hardware upgrades or incurring huge costs.

How I Built the FAISS Index

Building the index wasn't too complicated, but there were some things to keep in mind. Here's a quick breakdown of the steps I followed:

1. **Preparing the Embeddings**: First, I converted all the course descriptions, titles, and other text features into embeddings using the all-MiniLM-L6-v2 model. This resulted in a set of 384-dimensional vectors that captured the semantic meaning of the text.

2. **Creating the Index**: I used the IndexFlatL2 index type in FAISS, which calculates the Euclidean distance between vectors. Although this is one of the simplest types of indexes in FAISS, it works well for small datasets and gives exact results. Here's a snippet of the code I used:

```
import faiss
import numpy as np

# Assume embeddings is a numpy array of shape (59, 384)
index = faiss.IndexFlatL2(384)   # 384 is the dimension of the
embeddings
index.add(np.array(embeddings))# Adding the embeddings to the
index
```

3. **Testing for Speed and Accuracy**: I made sure to test the search speed and accuracy using a few sample queries. Even though the dataset was small, I wanted to confirm that the system could return the most relevant results quickly. For now, the search times were almost instantaneous, which is what I was aiming for.

Considerations for Scaling

While the IndexFlatL2 index type works well for a small number of entries, I'll need to switch to something more advanced, like IndexIVFFlat or IndexHNSW, as the dataset grows. These options allow for approximate search methods, which trade a little bit of accuracy for a huge boost in speed, making them better suited for larger datasets.

In the current setup, though, FAISS gave me the flexibility to start simple and have a plan for scaling up without completely overhauling the system. This approach lets me focus on building the rest of the search functionality while knowing that FAISS can handle the data growth when the time comes.

9. Search Functionality and LLM Integration

For this project, the main focus was on matching user queries with relevant courses by using embeddings and FAISS for similarity search. The initial setup worked well for identifying the most relevant results based on the semantic meaning of the query. To see if the search quality could be further improved, I decided to try integrating **Gemini** as an LLM.

How Gemini Was Integrated

The goal was to leverage Gemini to enhance the search results by:

- **Rephrasing or refining the user's query** to better capture the search intent.
- **Ranking or filtering the results** with additional context understanding.
- **Summarizing course descriptions** to help users quickly grasp the key points.

The integration was straightforward. I added Gemini into the search flow to process the user's input, aiming to provide refined results. I implemented code to handle this integration, including functions for query rephrasing and result re-ranking.

Despite integrating Gemini, I found that the changes in search quality were minimal:

- **Refined Queries Didn’t Make a Significant Difference**: Whether I used the original user query or the one rephrased by Gemini, the search results were almost identical. The embeddings and FAISS indexing were already capturing the semantic meaning well, and the added layer of LLM processing didn’t bring much improvement.
- **Summarization Had Little Impact**: Although Gemini could summarize course descriptions, the summaries didn’t add much value over the original details. Users still had to read the full descriptions to understand the course content properly.
- **Performance Drawbacks**: Integrating the LLM introduced more latency into the search process. This added delay wasn’t justified by the minimal improvement in search relevance, which led to a poorer user experience overall.

After evaluating the results, I decided not to fully remove the LLM-related code, even though the integration didn’t significantly improve the outcomes. The code remained in the project for potential future use, as it could still be helpful in scenarios where more complex natural language understanding might be required. However, for now, I chose to focus on the core embedding-based search functionality and left the LLM integration code without active use.

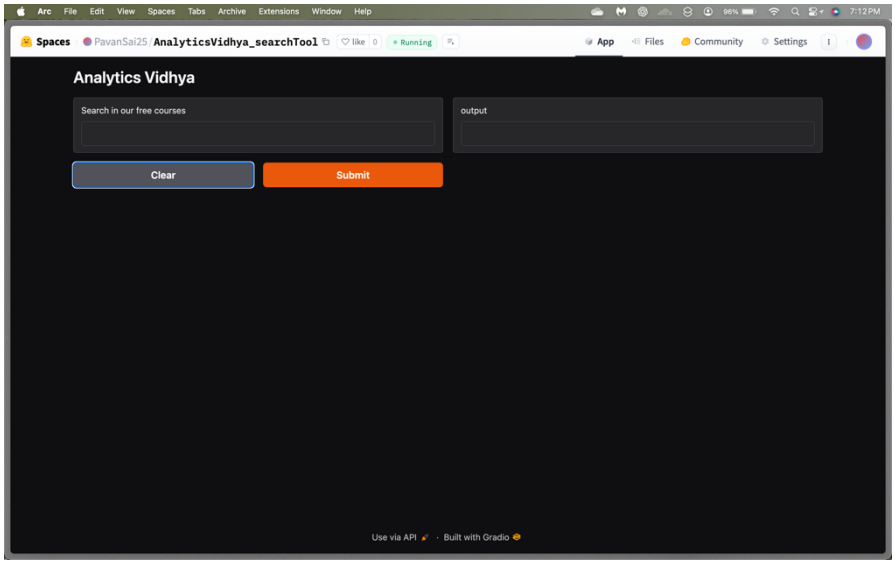
The experience showed that, while LLMs like Gemini can be powerful, they don’t always add value in cases where embeddings alone can achieve a strong level of search quality. This realization guided the decision to simplify the system and prioritize efficiency.

10. Deployment Approach

For deploying the smart search system, I chose to use **Gradio** for building the user interface and **Hugging Face Spaces** for hosting the application. This combination allowed me to create an interactive and accessible solution with minimal setup.

Why Gradio and Hugging Face?

- **Gradio** is a great tool for quickly building web-based interfaces for machine learning models. It made it easy to create a simple and user-friendly search interface where users could enter queries and see the search results displayed in real-time. The drag-and-drop nature of Gradio meant that I didn't need to spend a lot of time building the frontend from scratch.



- **Hugging Face Spaces** provided a convenient way to host the Gradio app. By deploying it on Hugging Face, I could leverage a free hosting solution that is well-integrated with machine learning projects. The setup was straightforward, and deploying updates was as simple as pushing changes to the associated GitHub repository.

Deployment Steps

Here's a brief rundown of how I set up the deployment:

1. **Developing the Gradio Interface**: I used Gradio to build a search interface that accepts user input and displays the top course results based on the FAISS search. The interface was designed to be minimal, with a text box for entering the search query and a panel to show the matching courses.
2. **Preparing the Hugging Face Space**: I created a new Space on Hugging Face and linked it to a GitHub repository where the project's code was stored. This way, every time I pushed a new commit, the changes would automatically reflect on the hosted app.

3. **Testing and Debugging**: After deploying, I tested the search system to ensure it worked smoothly and fixed any issues that came up. Given the simple nature of the Gradio interface, most adjustments were quick and easy.

Deploying on Hugging Face using Gradio not only made the project accessible but also allowed me to focus more on improving the search functionality rather than worrying about infrastructure or frontend development. The platform provided a good balance between ease of use and flexibility.

11. Testing and Evaluation

For testing the smart search system, I took a more straightforward approach. Instead of using metrics like precision or recall, I decided to manually test the system to get a feel for how well the search results matched user expectations.

Manual Testing Process
I ran a series of test queries across different topics and categories to see how relevant the returned courses were. For each search, I compared the top results to the expected outcomes based on the course content. This allowed me to get a practical sense of whether the system was finding relevant courses or if the results were off-target.

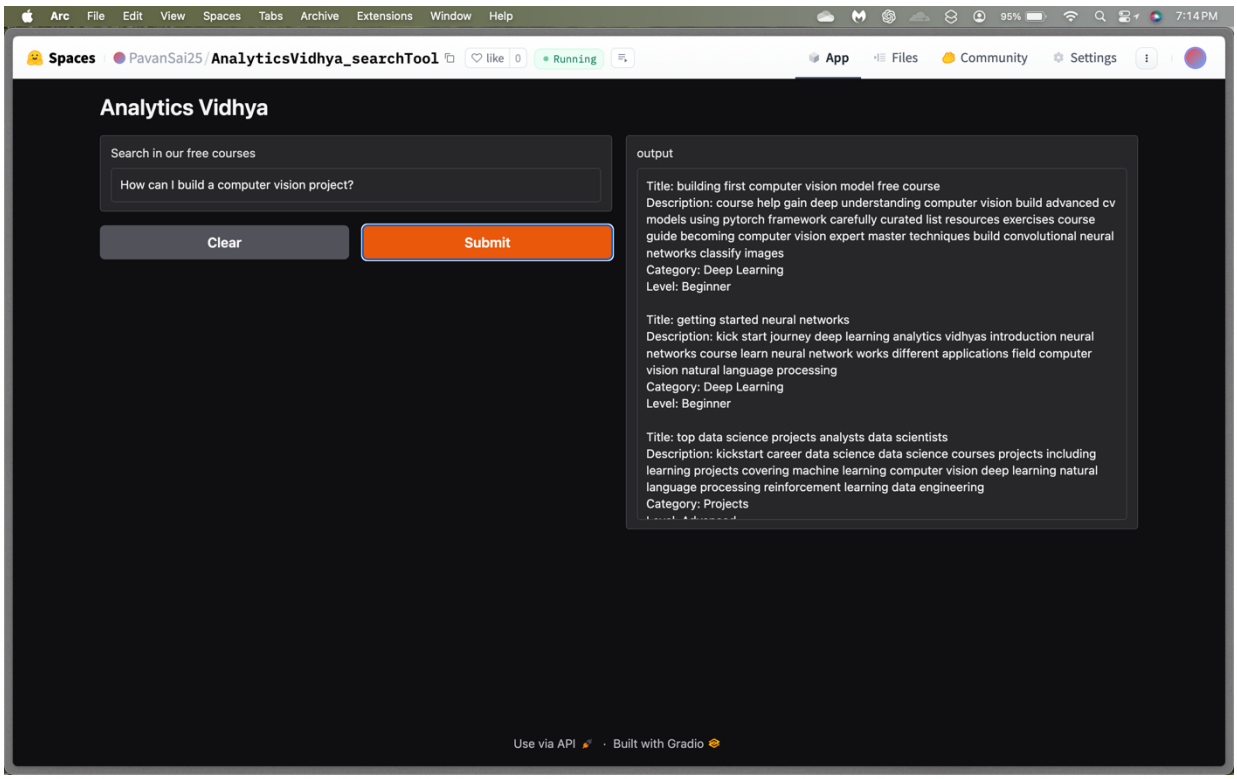
Observations and Results
Overall, the system performed reasonably well. The manual evaluation showed that the search results were often relevant, and in some cases, the ranking was better than I initially expected. For example:

- **General Queries** (e.g., "machine learning basics") returned courses that matched the topic fairly well, with the top results usually aligning with what a user would likely be looking for.
- **Specific Queries** (e.g., "advanced Python for data science") also yielded relevant results, though sometimes courses were ranked slightly out of the expected order.

While the results weren't perfect, they were definitely usable. There was a noticeable improvement in relevance when compared to a basic keyword search, thanks to the embeddings capturing semantic meaning.

Limitations and Challenges
Without using formal metrics like precision or recall, it's hard to quantify exactly how good the search performance is. The manual testing helped identify some patterns, but it also means the evaluation was somewhat subjective. I didn't

encounter any major issues during testing, but there's definitely room for further tuning to make the search even more accurate.



12. Conclusion and Future Work

The project focused on developing an intelligent search system for Analytics Vidhya courses, utilizing embeddings and language models to improve search relevance through semantic comprehension. The methodology incorporated the all-MiniLM-L6-v2 model for embedding generation and employed FAISS for effective similarity searches. Although there were challenges related to data collection and the integration of language models, the system was successfully launched on Hugging Face with Gradio, offering a user-friendly platform for course searches.

Achievements

Data Collection and Preprocessing: The project navigated challenges in web scraping by manually assembling a dataset of 59 courses, which included attributes such as course title, level, description, and category. Although some entries for paid courses were intentionally left incomplete, the preprocessing phase ensured that the dataset was clean and suitable for embedding generation.

Embedding-Based Search: Utilizing the all-MiniLM-L6-v2 model allowed for the capture of semantic relationships within the text, leading to search results that were more relevant than those produced by simple keyword matching.

FAISS Integration: The implementation of FAISS for similarity searches enabled the system to handle queries quickly and efficiently, even with the limited initial dataset.

Manual Evaluation of Results: Although formal precision-recall metrics were not applied, manual assessments suggested that the search results were fairly accurate and aligned with user expectations.

Limitations

Limited Dataset: The system was constrained by a dataset of only 59 courses, which restricted the diversity and depth of the search results.

Gemini LLM Integration: The integration of the Gemini language model did not yield significant enhancements in search relevance. The complexity added by the LLM was not warranted by the outcomes, resulting in its partial implementation in the final system.

Future Work

To further enhance the system, several potential areas for development can be considered:

Expanding the Dataset: Increasing the number of courses available, including both free and paid options, could enhance the diversity of search results. Additionally, integrating features such as course ratings and reviews would enrich the input for embeddings.

Utilizing More Advanced LLMs: Investigating other large language models, such as GPT-3.5 or GPT-4, or even fine-tuning models specifically tailored for educational content, could significantly enhance the search experience. These advanced models may improve the handling of complex queries, optimize result ranking, and offer personalized recommendations.

Incorporating Formal Evaluation Metrics: To more accurately assess search quality, employing metrics such as precision and recall would provide a more objective framework for evaluation and inform future enhancements.

Enhancing Real-Time Performance: For larger datasets, adopting a more advanced FAISS indexing strategy could boost search speed while preserving accuracy.

Introducing New Features: Adding functionalities like query expansion, synonym detection, or personalized search suggestions could enhance the system's versatility and user-friendliness.