Name: **Namver Ali Zaidi**

University: **Amity University, Gwalior**

Github Work Link:
https://github.com/Namverzaidi/Smart-Search-tool.git

Task: Here's what you need to do:

1. Browse through the free courses on the platform: Explore and collect data (e.g., course titles, descriptions, curriculum) from the free courses available on Analytics vidya platform.

2. Develop a smart search system: Build a search tool that quickly finds and suggests the most relevant courses using keywords and natural language queries. You should use Generative AI methods/tools like vector embeddings for querying and an LLM to get the results.

3. Deploy the tool on Huggingface Spaces: Once your search tool is ready, deploy it on Huggingface Spaces. You can use any framework (such as Gradio, Streamlit, or others)as long as it is publicly accessible via Huggingface Spaces.

4. Share the link for review: Provide us with the public link to your deployedsmart search tool for evaluation.

5. Share your approach: Share with us a document (e.g. docx, pdf, etc.) that explains how you have approached the problem, and the methodology behind the embedding model and LLM selection.

The more advanced and effective your search tool is, the better your chances of standing out.

Approach:

1. Web Scrapping:

- **Course Titles**:
  - **Reason**: The course title serves as the primary identifier for each course. It gives users a clear and concise understanding of what the course is about. Including this information is crucial for any search or filtering functionalities, as users often search for specific topics or titles.
- **Descriptions**:
  - **Reason**: Descriptions provide an overview of what the course covers, its objectives, and any prerequisites. They help users quickly assess whether the course aligns with their interests and learning goals. Detailed descriptions enhance the search experience by offering context around the course content.
- **Ratings**:
  - **Reason**: Ratings reflect user feedback and satisfaction levels with the course. This information is vital for potential learners to gauge the course's quality and effectiveness. Including ratings allows the smart search tool to prioritize highly rated courses, improving user experience by highlighting the most recommended options.
- **Duration**:
  - **Reason**: The duration indicates how long it will take to complete the course. This information helps users plan their learning schedule and manage their time effectively. For users with busy schedules, knowing the duration can influence their decision to enroll in a course, making it a critical data point for the search feature.
- **Level**:

- **Reason**: The skill level (e.g., Beginner, Intermediate) helps categorize courses based on the expertise required. This information is essential for users to find courses that match their current skill level. It also aids in filtering search results, ensuring users can easily locate appropriate courses without feeling overwhelmed by options that are too advanced or too basic.

 Links:

- **Reason**: Links provide direct access to the course details on the Analytics Vidhya website. Including this information is crucial for the search tool, as it allows users to quickly navigate to the course page for more in-depth information, such as enrollment options, curriculum, and instructor details. This enhances the usability of the smart search tool by facilitating easy access to relevant resources.

2. Explanation of the Next Step: Generating Course Description Embeddings

After successfully scraping the course data, the next step was to enhance the dataset by generating embeddings for the course descriptions. This step involved the following key activities:

1. **Loading the Scraped Data**:
    - The course data, which includes titles, descriptions, ratings, duration, level, and links, was loaded into a pandas DataFrame from the CSV file. This DataFrame serves as a structured format for further processing and analysis.
2. **Using Sentence Transformers**:
    - The **SentenceTransformer** model, specifically the all-MiniLM-L6-v2, was imported. This model is designed for creating high-quality embeddings for sentences or phrases. By using this pre-trained model, we can convert the textual descriptions into numerical vectors that capture their semantic meaning, which is crucial for natural language processing tasks like searching and similarity comparisons.
3. **Generating Embeddings**:
    - The course descriptions were extracted from the DataFrame and converted into a list. The SentenceTransformer model then encoded these descriptions, transforming them into embeddings. Each embedding is a fixed-size vector representation of the corresponding description, which captures the contextual relationships within the text. This process

allows the smart search tool to effectively compare and analyze descriptions based on their semantic content.

4. **Storing the Embeddings**:
    - The generated embeddings were added to the DataFrame as a new column, facilitating easy access for future steps, such as implementing the smart search functionality. This integration enhances the dataset, enabling advanced searching capabilities that consider the meaning behind the course descriptions.
5. **Saving the Updated DataFrame**:
    - Finally, the updated DataFrame, now containing the course descriptions and their corresponding embeddings, was saved to a pickle file. Pickle format is efficient for storing Python objects, making it suitable for quick retrieval during later stages of development. This step ensures that the embeddings can be reused without needing to regenerate them, saving time and computational resources.
6. **Confirmation Message**:
    - A success message was printed to indicate that the embeddings were generated and saved successfully. This provides feedback that the operation was completed without errors.

```
import pandas as pd
from sentence_transformers import SentenceTransformer
import numpy as np

df = pd.read_csv('output/analytics_vidhya_courses_with_ratings.csv')

model = SentenceTransformer('all-MiniLM-L6-v2')

course_descriptions = df['Description'].tolist()
description_embeddings = model.encode(course_descriptions)

# Store the embeddings into the DataFrame
df['Embeddings'] = list(description_embeddings)

# Save the DataFrame with embeddings for later use
df.to_pickle('output/courses_with_embeddings.pkl')
print("Embeddings generated and saved successfully!")
```

3. Explanation of the Search Functionality

In this step, we implemented a search functionality to allow users to find relevant courses based on their queries. The following outlines the approach taken in this process:

1. **Loading the Data with Embeddings**:
    - The DataFrame containing the course data and their corresponding embeddings was loaded from the pickle file. This DataFrame serves as the foundation for the search tool, providing access to all relevant course information.
2. **Model Initialization**:
    - The **SentenceTransformer** model (all-MiniLM-L6-v2) was initialized again, as it is necessary for generating embeddings from user queries.
3. **Search Function Definition**:
    - A function called search_courses was defined to handle the search process. This function takes a user-defined query string and an optional parameter top_n, which specifies how many of the most relevant courses to return.
4. **Generating Query Embedding**:
    - Within the search_courses function, the query string is encoded into an embedding using the same SentenceTransformer model. This converts the query into a vector representation that can be compared with the course description embeddings.
5. **Calculating Cosine Similarity**:
    - The **cosine similarity** between the query embedding and all course description embeddings is calculated using the cosine_similarity function from the sklearn.metrics.pairwise module. Cosine similarity measures the cosine of the angle between two vectors, providing a score that indicates how similar they are. A score closer to 1 means the vectors are very similar, while a score closer to 0 means they are less similar.
6. **Sorting and Retrieving Top Courses**:
    - The indices of the courses are sorted based on their similarity scores in descending order. The top N most relevant courses (as specified by the top_n parameter) are then retrieved from the DataFrame. This ensures that users receive the most pertinent results for their queries.

7. **Returning Relevant Information**:
    - The function returns a subset of the DataFrame containing the course title, description, rating, link, duration, and level for the top N courses. This makes it easy for users to view detailed information about the courses that match their interests.
8. **Example Query**:
    - An example search query was provided, such as "data science beginner course," to demonstrate how the function can be utilized. The results are printed, displaying the most relevant courses based on the user's query.

```
import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
from sentence_transformers import SentenceTransformer
# Load the DataFrame with embeddings
df = pd.read_pickle('output/courses_with_embeddings.pkl')

# Load the model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Function to perform a search query
def search_courses(query, top_n=5):
    # Generate the embedding for the query
    query_embedding = model.encode([query])

    # Calculate cosine similarity between the query and course descriptions
    similarities = cosine_similarity(query_embedding,
np.array(df['Embeddings'].tolist()))

    # Sort courses based on similarity score
    sorted_indices = np.argsort(similarities[0])[::-1]

    # Retrieve the top N most relevant courses
    top_courses = df.iloc[sorted_indices[:top_n]]

    return top_courses[['Course Title', 'Description', 'Rating', 'Link',
'Duration', 'Level']]

# Example search
query = "data science beginner course"
results = search_courses(query)
print(results)
```

4. Explanation of the Gradio Interface Implementation

In this step, we created a user-friendly interface for the smart search tool using **Gradio**. This allows users to interactively search for courses without needing to run any code. Here's a detailed breakdown of how the Gradio interface was implemented:

1. **Importing Necessary Libraries**:
    - The pandas library is used for handling data in a DataFrame format.
    - The gradio library is imported to create the web interface.
    - The custom search_courses function from the SmartSearchEngine module is imported to facilitate searching.
2. **Loading the Data**:
    - The previously saved DataFrame containing the course data and their embeddings is loaded from the pickle file. This DataFrame will be used to fetch relevant courses based on user queries.
3. **Defining the Search Function**:
    - A function named gradio_search is defined to serve as a bridge between the Gradio interface and the underlying search logic. This function takes a user query as input, calls the search_courses function with the query, and returns the results.
    - The results are converted into a dictionary format using to_dict(orient='records'), making them suitable for display in the Gradio interface.
4. **Creating the Gradio Interface**:
    - An instance of gr.Interface is created to define the user interface:
        - **fn**: Specifies the function to be called when the user submits a query (gradio_search).
        - **inputs**: Defines the input component, which is a text box allowing users to enter their search query. It has two lines and a placeholder text to guide users.
        - **outputs**: Sets the output format to display the search results as JSON, making it easy for users to read the information.
        - **title** and **description**: Provide a title and description for the interface to enhance user understanding.

5. **Launching the Interface**:
    - The iface.launch(share=True) method is called to start the Gradio interface. The share=True argument allows the interface to be accessible via a public link, enabling others to use the tool without needing to set up any local environment.

```
import pandas as pd
import gradio as gr
from SmartSearchEngine import search_courses

# Load the DataFrame with embeddings
df = pd.read_pickle('output/courses_with_embeddings.pkl')

# Define the search function to be used in the Gradio interface
def gradio_search(query):
    results = search_courses(query, top_n=5)
    return results.to_dict(orient='records')

# Create a Gradio interface
iface = gr.Interface(
    fn=gradio_search,
    inputs=gr.Textbox(lines=2, placeholder="Enter your search query here..."),
    outputs=gr.JSON(label="Search Results"),
    title="Smart Course Search Tool",
    description="Search for the most relevant courses on Analytics Vidhya"
)

# Launch the Gradio interface
iface.launch(share=True)
```

5. Screenshot of the Deployed application: