Approach to Building the Assessment Recommender

Objective

The goal of the project is to create a **recommendation system** that takes a **job description** or **natural language query** and returns relevant **assessment options** that match the provided criteria. The solution leverages **OpenAI embeddings** to understand the content of the assessments and match them with user queries based on their semantic similarity.

Key Components

1. **Data Source**:
    - The assessments data is stored in a CSV file (`assessments.csv`), which includes columns like **name**, **description**, **url**, **duration_text**, and **test_type**.
    - The dataset is processed to compute **embeddings** for each assessment using **OpenAI's Embedding model** (`text-embedding-ada-002`), which converts the textual information into a dense vector representation.
2. **User Interface (Streamlit)**:
    - The app is built using **Streamlit** to create a **user-friendly interface** where users can input a **job description** or **query**.
    - The app provides an input field for users to enter their requirements, and it returns the **best matching assessment** along with the **name**, **URL**, **duration**, and **similarity score**.
3. **Embedding and Similarity Matching**:
    - We compute embeddings for both the **query** entered by the user and the **assessments** stored in the dataset.
    - Using **cosine similarity**, we find the assessments that are most similar to the user's query. The assessment with the highest similarity score is returned as the recommendation.
4. **Caching**:

- The application leverages **Streamlit's caching** mechanism to store precomputed embeddings and avoid re-calculating them on each request, thereby speeding up the recommendation process.

Step-by-Step Explanation

1. Data Loading and Embedding Computation

The assessments data is loaded from a CSV file (`assessments.csv`). Each row contains the assessment's **name**, **description**, **duration_text**, **test_type**, and **URL**. For each assessment, we compute an **embedding** using **OpenAI's text embedding model**. The embeddings are stored in a pickle file (`assessments_with_emb.pkl`) for future use, avoiding the need to recompute embeddings on each session.

python
CopyEdit

```python
@st.cache_data
def load_data():
    # Load precomputed embeddings if available; else compute and cache
    if os.path.exists(PICKLE_PATH):
        df = pd.read_pickle(PICKLE_PATH)
    else:
        df = pd.read_csv(CSV_PATH)
        df[EMB_COL] = df.apply(lambda r: get_embedding(f"{r['name']}.
{r['description']}"), axis=1)
        df.to_pickle(PICKLE_PATH)
    return df
```

2. Getting Embeddings Using OpenAI API

The function `get_embedding()` calls **OpenAI's API** to compute the embedding for a given text (combination of name and description). The response is stored as a **NumPy array**.

python
CopyEdit

```python
def get_embedding(text: str) -> np.ndarray:
    """Call OpenAI to get an embedding for `text`."""
    resp = openai.Embedding.create(input=text, model=EMBEDDING_MODEL)
    return np.array(resp["data"][0]["embedding"], dtype=np.float32)
```

3. Query Processing and Finding Best Match

The query entered by the user is also embedded using the same method as the assessments. Then, we compute the **cosine similarity** between the query embedding and all precomputed assessment embeddings. The assessments with the highest similarity are returned.

python
CopyEdit
```
@st.cache_data
def find_best_match(query: str, df: pd.DataFrame, top_k: int = 1):
    """Embed the query, compute cosine sims against df, and return
top_k matches."""
    q_emb = get_embedding(query).reshape(1, -1)
    all_emb = np.vstack(df[EMB_COL].values)
    sims = cosine_similarity(q_emb, all_emb)[0]
    df2 = df.copy()
    df2["similarity"] = sims
    return df2.nlargest(top_k, "similarity")
```

4. User Interface with Streamlit

The **Streamlit interface** allows users to enter a job description or query, and then, upon clicking the "Get Recommendation" button, the system returns the most relevant assessment.

python
CopyEdit
```
def main():
    st.title("Assessment Recommender")
    st.write("Enter your assessment requirements to get the best
matching URL.")

    df = load_data()
    query = st.text_input("Enter your query:", "")

    if st.button("Get Recommendation") and query:
        with st.spinner("Fetching recommendation..."):
            best = find_best_match(query, df, top_k=1).iloc[0]
            st.markdown(f"**Name:** {best['name']}")
            st.markdown(f"**URL:** [{best['url']}]({best['url']})")
```

```
            if 'duration_text' in best:
                st.markdown(f"**Duration:** {best['duration_text']}")
            if 'test_type' in best:
                st.markdown(f"**Test Type:** {best['test_type']}")
            st.markdown(f"**Similarity Score:**
{best['similarity']:.4f}")
```

- **Text Input**: Users input their job description or query.
- **Recommendation**: The system provides the best matching assessment with a **similarity score**, **duration**, **test type**, and **URL**.

5. Running the Application

To run the application, execute the following code in your environment or Google Colab:

python
CopyEdit
```
if __name__ == "__main__":
    main()
```

API Endpoints and Structure

1. **Health Check Endpoint** (`/health`):
    - **Method**: `GET`
    - **Description**: Verifies the API is running and returns a basic status message.

Response Example:

json
CopyEdit
```
{
  "status": "up",
  "message": "API is running!"
}
```

-

2. **Recommendation Endpoint** (`/recommend`):

    - **Method**: `POST`

    - **Description**: Accepts a job description or query and returns the most relevant assessment.

Request Body Example:

json
CopyEdit
```
{
  "job_description": "Looking for a Python developer with experience in machine learning and data analysis."
}
```

-

Response Example:

json
CopyEdit
```
[
  {
    "title": "Python Skills Assessment",
    "duration": 60,
    "skills_assessed": ["Python", "Data Structures", "Machine Learning"]
  },
  {
    "title": "Machine Learning Algorithm Test",
    "duration": 45,
    "skills_assessed": ["Machine Learning", "Algorithms"]
  }
]
```

-

Conclusion

This solution implements an **Assessment Recommender** that leverages **OpenAI embeddings** and **Streamlit** for an interactive web application. The system processes natural language queries, computes their embeddings, and uses **cosine similarity** to match them with the most relevant assessments. The results are presented to the user via an intuitive Streamlit interface, and the application efficiently handles multiple user queries through caching.