Documentation - AI-Powered Job Search.

A **comprehensive documentation** covering the entire pipeline of our job search system. This project efficiently retrieves job listings using **FAISS-based vector search** and **Google Gemini AI** to **refine job recommendations**

Project Overview

The **AI-Powered Job Search System** is designed to **scrape job listings, store them efficiently, retrieve relevant matches, refine the results using an AI model (Google Gemini), and display them in an interactive UI**.

🔹 Steps Involved in the Project

1️⃣ **Scraping Data** → Extracting job listings from TimesJobs.

2️⃣ **Storing Data** → Converting scraped job details into vector embeddings and storing them in **FAISS**.

3️⃣ **Querying Jobs** → Retrieving job listings based on user queries and refining them using **Google Gemini AI**.

4️⃣ **User Interaction** → Displaying search results using **Streamlit UI**.

Project Structure

Plain Text

```
Evaluation Project/
|── src/                            # Source Code Directory
│   ├── scraper.py                   # Job data scraping script
│   ├── job_details.py                # Extracts job details from individual job listings
│   ├── store_job_data.py             # Converts job data to embeddings & stores in FAISS
│   ├── query_faiss.py                # Retrieves job results & refines using AI (Gemini)
│   ├── streamlit_app.py              # Frontend Streamlit UI for job search
│   ├── evaluate_ragas.py             # Evaluation script for RAGAS metrics
│   ├── utils.py                      # Utility functions (saving CSV, logging, etc.)
```

Individual File Explanations:

1. Scraper Module (`scraper.py`)

Purpose

This module scrapes job listings from **TimesJobs** and saves them in a structured **CSV format**.

Key Functionalities

- **Fetch Job Listings** → Extracts all jobs from multiple pages.
- **Extract Job Details** → Retrieves company name, skills, experience, salary, and more.
- **Store Data in CSV** → Saves all job details in `scraped_jobs.csv` .

This file is responsible for:

- Sending requests to the job listing page.
- Parsing the HTML content to extract job postings.
- Filtering jobs based on the posted date.
- Collecting basic job details (Company Name, Job Link).
- Calling `job_details.py` to extract detailed job information.
- Storing the extracted data in a structured format.
- Saves the extracted data using `save_to_csv()` .

Code Breakdown

1. **Importing Dependencies:**

```python
import requests
from bs4 import BeautifulSoup
from job_details import JobDetails
from utils import save_to_csv
```

- `requests` : Sends HTTP requests to fetch web pages.
- `BeautifulSoup` : Parses and extracts HTML content.
- `job_details` : Imports `JobDetails` class to get job details.
- `save_to_csv` : Saves extracted job data to a CSV file.

2. **Class `JobScraper`**

```python
class JobScraper:
    def __init__(self, base_url, headers):
        self.base_url = base_url
        self.headers = headers

```

- **`base_url`** : Stores the main job listing page URL.
- **`headers`** : Sets a User-Agent to avoid bot detection

3. **`fetch_jobs()` – Fetches the Job Listing Page**

```python
def fetch_jobs(self, page=1):
    url = f"{self.base_url}?page={page}"
    response = requests.get(url, headers=self.headers)
    return response.text if response.status_code == 200 else None
```

- Constructs the **URL with pagination** (`page=1` , `page=2` , etc.).
- Sends an HTTP request using `requests.get()` .
- Returns the HTML content if the request is successful.

4. **`extract_jobs()` – Extracts Jobs from HTML**

Python

```python
def extract_jobs(self, html):
    soup = BeautifulSoup(html, "html.parser")
    job_elements = soup.find_all("div", class_="job-listing")

    jobs = []
    for job in job_elements:
        company = job.find("h3", class_="company-name").text.strip()
        job_link = job.find("a", class_="job-link")["href"]
        posted_date = job.find("span", class_="posted-date").text.strip()

        # Call job_details.py to get detailed info
        details = JobDetails.get_job_details(job_link)

        jobs.append({
            "Company Name": company,
            "Required Skills": details["skills"],
            "Experience": details["experience"],
            "Salary": details["salary"],
            "Location": details["location"],
            "More Info": job_link
        })
    return jobs

```

- **Parses HTML** to find job postings (`div.job-listing`).
- Extracts **Company Name, Job Link, and Posted Date**.
- Calls `JobDetails.get_job_details(job_link)` to get more details.
- Returns the extracted job data as a list of dictionaries.

5. **`main()` – Runs the Scraper**

Python

```python
if __name__ == "__main__":
    scraper = JobScraper(base_url="https://example.com/jobs", headers={"User-Agent": "Mozilla/5.0"})

    all_jobs = []
    for page in range(a, b):
        html = scraper.fetch_jobs(page)
        if html:
            jobs = scraper.extract_jobs(html)
            all_jobs.extend(jobs)

    save_to_csv(all_jobs, "scraped_jobs.csv")
    print("✅ Job data saved successfully!")

```

- Initializes the `JobScraper` with the **job site URL and headers**.
- Loops through **multiple pages (a to b)** to fetch job listings.
- Extracted jobs are **saved to a CSV file** using `save_to_csv()`.

2.Job Details Extraction (`job_details.py`)

This file is responsible for:

- Visiting each job's detailed page.
- Extracting **Required Skills, Experience, Salary, and Location**.

Key Sections

2.1 `get_job_details(url)`

- Sends a request to the job details page.
- Parses the page to extract:
  - Required Skills
  - Experience
  - Salary
  - Location
- Returns the extracted details.

1. **Importing Dependencies**

Python
```python
import requests
from bs4 import BeautifulSoup
```

- `requests`: Fetches the job details page.
- `BeautifulSoup`: Extracts data from the page.

2. **Class `JobDetails`**

Python

```python
class JobDetails:
    @staticmethod
    def get_job_details(url):
        response = requests.get(url, headers={"User-Agent": "Mozilla/5.0"})
        if response.status_code != 200:
            return {"skills": "N/A", "experience": "N/A", "salary": "N/A", "location": "N/A"}
```

- **Sends a request** to the job details page.
- If the request **fails (not 200)**, returns `"N/A"` for all fields.

3. **Extracting Job Details**

Python

```python
soup = BeautifulSoup(response.text, "html.parser")
skills = soup.find("span", class_="skills").text.strip()
experience = soup.find("span", class_="experience").text.strip()
salary = soup.find("span", class_="salary").text.strip()
location = soup.find("span", class_="location").text.strip()

return {
    "skills": skills,
    "experience": experience,
    "salary": salary,
    "location": location
}

```

- Parses the page to extract **skills, experience, salary, and location**.
- Returns a dictionary containing the extracted information.

---

3. Storing Job Data (`store_job_data.py`)

Purpose

Converts scraped job data into **vector embeddings** and stores it in **FAISS** for efficient retrieval.

Key Functionalities

- **Convert job text into embeddings using Hugging Face model**
- **Store job embeddings in FAISS**
- **Save metadata (company, skills, experience) in JSON for quick lookup**

Key Steps

1. **Read job data from `scraped_jobs.csv`**
2. **Convert job descriptions into embeddings** using a **sentence-transformer model**
3. **Store the embeddings in a FAISS index**
4. **Save the FAISS index and job metadata for retrieval**

1. **Importing Dependencies**

Python
```python
import os
import faiss
import numpy as np
import pandas as pd
import json
from transformers import AutoTokenizer, AutoModel
import torch
```

- `faiss` → Stores job embeddings in a **vector database** for fast retrieval
- `numpy` → Handles embedding arrays
- `pandas` → Reads job data from CSV
- `json` → Stores job metadata for reference
- `transformers & torch` → Loads **Hugging Face sentence-transformer model** for embedding generation

2. **Load Job Data from CSV**

Python
```python
CSV_FILE = "scraped_jobs.csv"

if not os.path.exists(CSV_FILE):
    raise FileNotFoundError(f"⚠️ CSV file '{CSV_FILE}' not found! Run the scraper first.")

df = pd.read_csv(CSV_FILE)
print(f"✅ Loaded {len(df)} job listings from CSV.")
```

- Checks if `scraped_jobs.csv` **exists**
- Reads job data into a **Pandas DataFrame (`df`)**
- Prints the **total number of jobs loaded**

3. Load Hugging Face Embedding Model

Python

```python
EMBEDDING_MODEL = "sentence-transformers/all-MiniLM-L6-v2"

tokenizer = AutoTokenizer.from_pretrained(EMBEDDING_MODEL)
model = AutoModel.from_pretrained(EMBEDDING_MODEL)
```

- Loads the **"all-MiniLM-L6-v2"** model
- This model converts **job descriptions into numerical embeddings**

4. Function: Convert Text to Embedding

Python

```python
def get_embedding(text):
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
    with torch.no_grad():
        outputs = model(**inputs)
    return outputs.last_hidden_state[:, 0, :].numpy().astype("float32")
```

- Converts **text into a numerical vector** (embedding)
- Uses **Hugging Face tokenizer and model**
- Extracts the **first token's embedding** (sentence representation)
- Returns a **NumPy array** in **float32 format** (FAISS requirement)

5. **Process Job Data & Generate Embeddings**

Python

```python
job_data = []
embeddings = []

for _, row in df.iterrows():
    job_desc = f"{row['Company Name']} - {row['Required Skills']} - {row['Experience']} years - {row['Location']}"
    embedding = get_embedding(job_desc)

    job_data.append({
        "Company Name": row["Company Name"],
        "Required Skills": row["Required Skills"],
        "Experience": row["Experience"],
        "Salary": row["Salary"],
        "Location": row["Location"],
        "More Info": row["More Info"]
    })
    embeddings.append(embedding)
```

- **Loops through each job listing**
- Constructs a **job description string**
- Converts the **job description into an embedding**
- Stores job details in a `job_data` list (for metadata storage)

- Stores **embeddings** in a list

6. **Convert Embeddings to FAISS Format**

```python
embeddings = np.vstack(embeddings)  # Convert list of embeddings to NumPy array
dimension = embeddings.shape[1]  # Get the embedding vector size
index = faiss.IndexFlatL2(dimension)  # Create FAISS L2 (Euclidean) index
index.add(embeddings)  # Add job embeddings to the FAISS index
print(f"✅ FAISS index created with {len(embeddings)} job listings.")
```

- Converts the **list of embeddings into a NumPy array**
- Retrieves **vector dimensions** from the embeddings
- Creates a **FAISS index** using `IndexFlatL2()` (Euclidean distance)
- Adds **all job embeddings** into the FAISS index

7. **Save FAISS Index and Job Metadata**

```python
faiss.write_index(index, "job_index.faiss")

with open("job_data.json", "w") as f:
    json.dump(job_data, f, indent=4)

print("✅ FAISS index and job data saved successfully!")

```

- Saves the **FAISS index to `job_index.faiss`**
- Saves the **job metadata to `job_data.json`**
- Prints a success message

---

4. Query Processing (`query_faiss.py`)

Overview of the Query Process

1. **Load the FAISS index and job data**: Retrieves stored job listings and their embeddings.
2. **Embed the user query**: Converts the search query into a vector representation.
3. **Search FAISS for relevant jobs**: Finds similar jobs based on **embedding similarity**.
4. **Refine results with Google Gemini**: The retrieved jobs are **formatted and ranked** using an LLM.
5. **Return the best job matches**: The top job listings are returned in a structured format.

Code Breakdown

1.Load Environment Variables

Python

```python
from dotenv import load_dotenv
import os

load_dotenv()
API_KEY = os.getenv("GEMINI_API_KEY")
if not API_KEY:
    raise ValueError("Google Gemini API key not found! Set GEMINI_API_KEY in your environment.")
```

- Loads environment variables using `dotenv` .
- Retrieves the **Google Gemini API key** from the environment.
- Throws an error if the API key is missing, ensuring the script doesn't proceed without authentication.

2.Load FAISS Index & Job Data

Python

```python
import faiss
import json

try:
    index = faiss.read_index("job_index.faiss")
    print("✅ FAISS index loaded successfully!")
except Exception as e:
    raise ValueError(f"Error loading FAISS index: {e}")

try:
    with open("job_data.json", "r") as f:
        job_data = json.load(f)
    print(f"✅ Loaded {len(job_data)} job listings from JSON.")
except Exception as e:
    raise ValueError(f"Error loading job data JSON: {e}")
```

- Loads the **FAISS index**, which stores job embeddings for fast similarity search.
- Loads **job_data.json**, which contains job details.
- Handles errors to prevent crashes if the files are missing or corrupt.

3.Load Hugging Face Model for Embeddings

Python

```python
from transformers import AutoTokenizer, AutoModel
import torch

EMBEDDING_MODEL = "sentence-transformers/all-MiniLM-L6-v2"
tokenizer = AutoTokenizer.from_pretrained(EMBEDDING_MODEL)
model = AutoModel.from_pretrained(EMBEDDING_MODEL)
```

- Loads the **sentence-transformers/all-MiniLM-L6-v2** model from Hugging Face.
- Initializes the **tokenizer** to process text inputs.
- Loads the **pre-trained model**, which converts text into numerical embeddings.

4.Generate Embeddings for User Query

Python

```python
def get_embedding(text):
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
    with torch.no_grad():
        outputs = model(**inputs)
    return outputs.last_hidden_state[:, 0, :].numpy().astype("float32")
```

- Tokenizes the input text using **Hugging Face tokenizer**.
- Passes the tokenized input through the model to get embeddings.
- Extracts the first token's embedding (`[:, 0, :]`), representing the sentence meaning.
- Converts the embedding into a **NumPy array** for compatibility with FAISS.

5. Search FAISS for Relevant Jobs

Python

```python
def search_jobs(query, k=10):
    query_embedding = get_embedding(query).reshape(1, -1) # Ensure 2D format

    distances, indices = index.search(query_embedding, k)
    matched_jobs = [job_data[idx] for idx in indices[0] if idx >= 0 and idx < len(job_data)]

    if not matched_jobs:
        return "⚠️ No jobs found. Kindly change your query and try again."
```

- Converts the **user's query** into an embedding using `get_embedding()` .
- Searches the FAISS index for **similar job listings** based on embedding similarity.
- Retrieves job details from `job_data.json` using the **matching indices**.

- Returns a **warning message** if no jobs are found.

6. Format Job Listings for Google Gemini

```python
job_context = "\n\n".join(
    [
        f"🏢 **Company:** {job['company']}\n"
        f"💼 **Required Skills:** {job['requiredSkills']}\n"
        f"📅 **Experience:** {job['experience']} years\n"
        f"📍 **Location:** {job['location']}\n"
        f"🔗 **More Info:** {job['moreInfo']}"
        for job in matched_jobs
    ]
)
```

- Creates a formatted string containing **company name, skills, experience, location, and job link**.
- Uses `\n\n` to separate job entries for better readability.
- Ensures that **Google Gemini receives structured input** for improved response quality.

7.Generate AI-Powered Response with Google Gemini

```python
prompt = f"""
You are an AI job assistant. Based on the given user query, refine the provided job listings and show the **5 most relevant** ones.

**Rules:**
- Select the most relevant jobs based **only on the provided listings**.
- Display jobs that match the required experience level.
- Do not add or modify information—only filter and reformat.
- The output should follow this structure:

  "**[Company Name]**
  They are looking for candidates with **[Required Skills]** and require **[Experience] years** of experience.
  The job is based in **[Location]**.
  If you're interested, click below to apply: [More Info]"

**User Query:** {query}

**Job Listings:**
{job_context}
"""
```

- Creates a structured **prompt for Gemini** to ensure high-quality job recommendations.
- Enforces **strict rules** to prevent hallucinations or modifications to job listings.
- Encourages Gemini to return **clear, easy-to-read job descriptions**.

8. Send Query to Google Gemini

Python

```python
import google.generativeai as genai

genai.configure(api_key=API_KEY)

try:
    response = genai.GenerativeModel(MODEL_NAME).generate_content(prompt)
    refined_results = response.text.strip() if hasattr(response, 'text') else "⚠️ No response from Gemini."
except Exception as e:
    refined_results = f"Error in Gemini API: {e}"
```

- Configures the **Google Gemini API** with the provided key.
- Sends the formatted job listings to **Gemini for ranking and refinement**.
- Extracts and **cleans the response**, ensuring a smooth user experience.
- **Handles API errors** gracefully, preventing crashes if the request fails.

---

5. Frontend UI (`streamlit_app.py`)

Key Features Implemented:

1. A **search bar** for users to input job-related queries.
2. Integration with the **FAISS-based job search pipeline**.
3. Display of **AI-refined job listings** retrieved using the **Google Gemini API**.
4. Clean and structured **UI/UX** for a seamless experience.

1.Setting Up Streamlit and Loading Dependencies

Python

```python
import streamlit as st
from query_faiss import search_jobs  # Import FAISS + Gemini function

```

1. **Streamlit** is used to create an interactive web-based UI for job searching.
2. The `query_faiss.py` file contains the function `search_jobs()`, which integrates FAISS and

Google Gemini to retrieve job listings.

3. This function takes a user query, searches for relevant jobs, and refines the results before displaying them.
4. By importing it here, we can easily use it inside our Streamlit app.

2. Creating the Streamlit UI

```python
st.title("🔍 AI-Powered Job Search")
st.write("Enter a job query below and get AI-refined job listings!")
```

1. The `st.title()` function displays the **main heading** of the application.
2. `st.write()` provides a **short description**, guiding users on how to use the app.
3. This ensures users understand that they need to input a search query to find job listings.
4. The interface is designed to be **minimalistic and user-friendly** for better accessibility.

3. Capturing User Input

```python
query = st.text_input("💼 Job Search Query:", "")
```

1. `st.text_input()` creates a **text box** where users can enter their job search queries.
2. The placeholder text `"💼 Job Search Query:"` provides a **clear indication** of what users should type.
3. The input is stored in the `query` variable for further processing.
4. If the input is empty, the app will display a **warning message** later.

4. Handling Search Button Click

```python
if st.button("Search Jobs"):
    if query:
        st.info("🔄 Searching for the best job matches...")
```

1. `st.button("Search Jobs")` creates a **clickable button** to start the search.
2. If the button is clicked and the query is **not empty**, a loading message (`st.info()`) is displayed.
3. The loading message `"🔄 Searching for the best job matches..."` informs users that the system is processing their request.
4. This enhances the **user experience** by providing feedback on the search process.

5. Fetching AI-Refined Job Listings

```python
refined_results = search_jobs(query, k=10)
```

1. Calls the `search_jobs()` function, which searches for jobs using **FAISS and Gemini AI**.
2. The query is converted into an **embedding** and compared with stored job embeddings.
3. The top **10 relevant job listings** (`k=10`) are retrieved from the FAISS index.
4. The results are then **refined** using the Gemini AI model before being displayed.

6. Displaying Search Results

```python
if refined_results:
    st.subheader("🎯 AI-Refined Job Listings:")

    jobs = refined_results.split("\n\n")  # Assumes jobs are separated by double newlines

    for idx, job in enumerate(jobs, 1):
        lines = job.splitlines()
        if len(lines) > 1:
            company_line = lines[0].strip()  # First line is company name
            details = "\n".join(lines[1:])  # Remaining job details

            # ✅ Display company name in bold + "is hiring!"
            st.markdown(f"<h3><b>{idx}. {company_line} is hiring!</b></h3>", unsafe_allow_html=True)

            # ✅ Display job details below with spacing
            st.write(details)

        st.markdown("<hr style='border: 1px solid #ccc; margin: 20px 0;'>", unsafe_allow_html=True)  # Add spacing

```

1. If job listings are found, a **subheader** " 🎯 `AI-Refined Job Listings:"` is displayed.
2. The AI response is assumed to be **formatted with double newlines** between job listings.
3. Each job is processed individually, with the **company name in bold** and details shown below.
4. The `st.markdown()` function ensures that job listings are **well-structured and visually distinct**.
5. A horizontal line (`<hr>`) is added between job listings for **better readability**.

7. Handling No Results & Empty Queries

Python
```python
else:
    st.warning("❌ No relevant jobs found.")
```

1. If no jobs are found, Streamlit shows a **warning message** "❌ No relevant jobs found."
2. This informs the user that their search query might not match any stored job listings.
3. Users can refine their queries and try again for better results.
4. This improves the **overall user experience** by handling unsuccessful searches gracefully.

Python
```python
else:
    st.warning("⚠️ Please enter a job query.")
```

1. If the search button is clicked **without entering a query**, a warning is displayed.
2. This prevents unnecessary API calls and ensures users input a **valid search query**.
3. Providing **clear feedback** helps users correct mistakes before submitting.
4. This enhances the app's usability and reduces potential confusion.