**Automate Your Job Hunt with Gen AI and Selenium**

Applying for jobs is a long and complicated process that can become tedious and mentally exhausting. More than anything, it is repetitive and boring. However, this very predictability makes it a well-defined use case for automation. And if our resumes are being scanned by AI, why not use AI to apply for jobs as well?

We can approach the project in two parts,

1. Handling the web elements and automation
2. A Query Engine to handle all the questions in the application.

To handle web elements, we can use the Selenium package with python. Selenium offers straightforward and intuitive functions that allow you to search for and interact with web elements seamlessly.

The query engine, on the other hand, is more complex due to the dynamic nature of job applications—each one presents new and unique questions. It is impossible to anticipate all potential queries in advance. So, we need a system that can adapt to the user’s profile, answering questions accurately and concisely based on the available data. This is where the **RAG (Retrieval-Augmented Generation)** system excels, as it is purpose-built for scenarios like this, allowing for focused, context-aware responses in real time.

**Project Overview**

[Insert flow diagram here]

**Prerequisites and Setup:**

* Python3, pip package handler, Jupyter
* **Ollama** (for local inference). Download from [here](https://ollama.ai/). If you plan to use GPT-4, you can skip this step.
  + Run ollama get phi3 model detail

[insert code snippet]

* + If available, use a CUDA-supported GPU (optional).
* **Hugging Face** and **GPT-4 API keys**. [Get Hugging Face](https://huggingface.co/), and GPT-4 requires paid access [here](https://openai.com/).
* **Create a virtual environment** (recommended), as the extensive list of installations might conflict with your current Python environment.
* **[ create venv]**
* Install the packages from the requirements.txt file.
* [install packages using pip]
* I used **Edge** because it’s light and fast. However, you can choose another WebDriver. Download it [here](https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/).
* [ driver]
* Write functions to handle JSON and Excel files.
* [code snippet]
* Ensure to include all file paths.
* [ file paths]

**Selenium: Applying for Jobs**

* Open LinkedIn using the WebDriver, navigate to the login page, and enter your email and password.

[login syntax]

* Navigate to the job search page, input the role and location, and click search.
* [job search]
* Once the search results appear, apply filters. If the search yields no results, reset the filters.
* [apply filters]
* Lastly, apply the **Easy Apply** filter.
* [ easy apply]

**Automating Job Applications:**

* Write functions to handle each form element (e.g., text fields, radio buttons, dropdowns). Pass the element and the required input to the function to automate form filling.
* [syntax for radio and drop down inputs]
* Create additional functions to open, submit, and close job applications.
* [function for submit, close]
* Implement an iterator to iterate through job search results and pages. Ensure it checks whether you have already applied to a particular job.
* [iterator code]
* Create a job\_apply function to use the above handlers to open, fill, and submit applications.
* [apply function]
* Extract job details and write them to a file.
* In case of errors, allow space for manual input and retries.
* [manual retry]

**Query Engine**

After handling form elements and automating the job search process, we move to the query engine, which involves **Local Cache with Hashing** and **RAG (Retrieval-Augmented Generation)**.

**Hashing, Fuzzy Matching, and Local Cache:**

* Although advanced LLMs exist, nothing beats a good old O(1) search. All question-answer pairs encountered in job applications are stored as key-value pairs in a JSON file.
* If the same question appears again, it is accessed via a simple hash search.
* For questions with similar phrasing, fuzzy matching (90% threshold) is used to find the closest match in the keys and retrieve the appropriate answer.
* If hashing and fuzzy matching fail, the system defaults to using **RAG**.
* [ add code for the reading json and query engine]

**Agentic RAG with LlamaIndex**

RAG stands for **Retrieval-Augmented Generation**, a method that combines information retrieval with text generation. When a query is made, the RAG system retrieves relevant information (from a resume or related documents), which is passed along with the query to a language model for generating responses.

**LlamaIndex** is an open-source framework used to build LLM-based applications by providing tools for data ingestion, indexing, and querying. By default, LlamaIndex uses GPT-3.5 as the default LLM.

**Data Preparation:**

Since resumes are packed with concise information, LLMs might overlook some details.

So we take the resume, feed it to GPT3.5 and generate an elaborated version of the resume which has more detailed descriptions of the current resume and would be easier to read for LLM models.

[ add gpt 3.5 read and write code here]

**Data Ingestion**:

* Ingest data by giving the docs folder path to **SimpleDirectoryReader** package from LlamaIndex.
* Split the documents into nodes of appropriate sizes.

[Explain why splitting documents into nodes is important]

* The nodes are converted into embeddings, and a **VectorDB index** is created using an embedding model.
* [ insert code snippet for getting the data, splitting and indexing]

**Phi3-mini:**

* Skip this step if using GPT-4.
* **Phi3-mini** is part of Microsoft’s **Phi-3 family**, a series of **small language models (SLMs)** designed to be efficient and lightweight while maintaining high quality.
* I implemented the RAG system locally using Phi3-mini (1.5B parameters). Using a resource-heavy model like GPT-4 for every use case is overkill, and I believe case-specific SLMs are the future.
* The embedding model used is the **BAAI/bge-small-en-v1.5**.
* [ insert code snippet for getting the LLm , embedding model, creating the vector index]

After downloading the required Phi3 model, assign it and the embedding model to the LlamaIndex query engine. The engine retrieves relevant documents and passes them as context to the prompt. For this, **prompt engineering** is crucial to get accurate results.

[prompt engineering]

**GPT-3.5:**

While **Phi3-mini** is efficient for Q&A-based queries, it underperforms for tasks like generating long summaries or cover letters. For these tasks, I switched to **GPT-3.5**. The setup is similar to the Phi3-mini configuration, except for replacing the model and embedding model. Both systems share the same data source.

[ insert llm model and embedding code]

[explain the difference in both indices]

[insert prompt engineering]

**Final Query Engine:**

The **Agentic RAG** model improves decision-making capabilities using agents, enhancing the RAG system’s ability to handle more complex queries. In this case, I implemented the **Router function** in LlamaIndex.

To use the router, package both the Phi3 and GPT-3.5 query engines into tools accessible by the router. Then, create a router query engine that forwards the user query to the appropriate tool based on the query type.

[ syntax for the router query engine]

Here we have two tools, vector tool for short concise, QnA and summary tool for longer Summary based questions or cover letters.

I also tried few shot examples to make the queries more precise but the drop in performance was not worth the increase in quality of the response.

**Persistent Storage:**

Since index creation is time-consuming, I stored the indices locally, allowing faster access when the code runs again.

Both vector tools and summary tools have their own indices

[ snippet for vector index]

[snippet for summary index]

**FastAPI and Uvicorn:**

The entire RAG system is wrapped into an API using the **FastAPI framework** and hosted locally using **Uvicorn**. This API can then be accessed by any local machine.

* [API creation instructions]
* [API hosting details]
* [Accessing the API from the final code]

**Running the Code:**

* Start the API.
* Run the Selenium script.
* [add code snippet for the iterator]
* The jobs applied for should be populated in the Excel file.
* [add video]