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

Applying for jobs is tedious and sometimes, mentally exhausting. More than anything, it’s the repetitiveness that gets to us. 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?

This project approach can be broadly divided into two parts:

1. Handling the web elements and automation
2. A dynamic 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 learn the user’s profile and can generate answers for any new questions encountered. This is where the **RAG (Retrieval-Augmented Generation)** systems excel, as they’re purpose built for use cases like this.

**Project Overview**

Here’s a flowchart that provides a high-level overview of the project. As mentioned earlier, web interactions are managed by Selenium. The query engine is implemented using a combination of hashing and RAG (Retrieval-Augmented Generation). The query engine is then packaged into an API using FastAPI, running on a Uvicorn server.

[Insert flow diagram here]

**Prerequisites:**

• Python 3, pip package manager, and Jupyter Notebook.

• Ollama (for local inference). You can download it from here. If you plan to use GPT-4, you can skip this step.

In a command line, run: Ollama run phi3:3.8b-mini-4k-instruct-q4\_K\_M

For local inference, it is recommended to use a device with a CUDA-supported GPU.

• Hugging Face and OpenAI API keys (Note: OpenAI requires credits).

• Create a virtual environment (recommended) to avoid conflicts with your current Python environment, as there are many dependencies to install.

• WebDriver: I used Edge’s WebDriver, but any popular browser’s WebDriver will work.

**Code and Instructions:**

Now that the setup is complete, let's walk through the code and implementation of this project step by step.

1. **Web Interaction with Selenium:**

**Logging in and Job Search**

All functionalities are implemented as functions, which helps keep the code clean, modular, and avoids redundancy.

Start by opening LinkedIn's sign-in page and logging in with your credentials. This step is straightforward.

[insert code snippet]

Once logged in, navigate to the job search page. Input the job title and location, then click search. •

[job search]

After the search results appear, apply filters to refine the results to better match your profile. If the filters yield no results, reset them. •

[apply filters]

Lastly, apply the "Easy Apply" filter. Since this project focuses on using Easy Apply for job applications, this filter is essential. •

[Easy Apply]

**Locating and Handling Application Form Elements:**

Application forms contain various types of elements, and the only way to identify them all is by reviewing several applications. After examining the forms, the types of form elements can be categorized as radio buttons, checkboxes, single-line text fields, multi-line text fields, autofill text fields, and dropdowns.

Handling these form elements is key to answering the questions on the application. Each element is managed by specific functions tailored to its requirements. These functions locate the element, retrieve the corresponding answer from the query module, and insert the answer in the correct format.

[insert syntax]

Next, define an **Apply** function that locates these elements on the application, then calls the appropriate handlers to manage them. Also, incorporate loggers to track the details of the companies applied to.

[insert code snippet]

Additionally, create ad-hoc functions to open, submit, and close job applications. • [submit and close functions]

Lastly, implement an iterator to loop through the search results and navigate to the next page once the applications on the current page are complete. •

[iterator code]

With the iterator in place, the code should be able to navigate through the search results and handle the entire job application process for each job. However, it still lacks the answers to fill in the application forms.

1. **Handling Application Form Response:**

Although many of the questions in job applications are repetitive, they can't be generalized across all applications, as each job posting has its own unique requirements and questions.

This can be handled by a combination of local cache for frequently repeated questions and a RAG system to answer questions that haven’t previously encountered.

**Local Cache with Hashing And Fuzzy Matching:**

* While advanced LLMs are powerful, nothing beats the simplicity of an O(1) search. Since most repetitive questions have identical phrasing, they can be stored as key-value pairs in JSON records and accessed via a simple hash search. For questions with similar but not exact phrasing, fuzzy matching (minimum edit distance) can be applied to the list of keys in the JSON, using a 90% match threshold to find existing answers. Although slower than a direct hash search, this approach is still an algorithmic search and remains faster than using a Language Model. [ add code snippet]
* New questions are added to these records after each application and the updated json file is ingested by the RAG setup.

**Agentic RAG with LlamaIndex**

If algorithmic search yield no results, it implies the question may not be present in the json records and we handover the query to the RAG system.

RAG, or Retrieval-Augmented Generation, is a method that integrates information retrieval with text generation. When a query is made, the RAG system retrieves relevant data from a pre-indexed source. This retrieved information serves as context and is then combined with the query and passed to a language model, which generates the response.

For this use case, the RAG system can be implemented using LlamaIndex, an open-source framework designed for building LLM-based applications. It provides tools for data ingestion, indexing, and querying. By default, LlamaIndex utilizes GPT-3.5 as the default language model.

Here’s the walkthough of the RAG workflow:

**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**:

The input data folder consists of user’s resume in a pdf format, rewritten elaborated resume in a txtfile format and the json file that has the question and answers data of previously occurred questions.

* LlamaIndex offers various tools for ingesting data from different formats. The **SimpleDirectoryReader** package handles most file formats by taking files from a directory and loading them as documents.
* These documents are then split into smaller chunks or nodes to facilitate indexing and improve readability by the model. Additionally, since LLMs have token size limits, splitting the documents ensures the inputs remain within those limits. In this case, the documents are split into chunks with a token size of 1024.
* Once the documents are split into nodes, they are converted into embeddings. An indexed dataset is then created using a **VectorDB index**, which allows the system to retrieve relevant chunks based on a specific query.
* [ insert code snippet for getting the data, splitting and indexing]

Agentic RAG:

I wanted to implement the RAG system entirely on my local device. So I was looking for models with small number of parameters phi3-mini-instruct for this use case. Which is a small Language model with around 3.8 Billion parameters when compared to a large language model like GPT4 which is estimated to have 1.8 trillion parameters. The model performed well and was on par with models 10 times its size. It did really well with short Question and Answers but was failing with long form summary-based questions. For these tasks we can send the query to GPT 3.5. these two modes can be used together using an Agentic Rag system.

An Agentic RAG model refers to a RAG system that incorporates an agent capable of making automated decisions. These decisions guide the model's behavior during the retrieval and generation process. In this use case, Agentic RAG is used to determine whether to route the query to the **phi-3-mini model** or the **GPT-3.5 model**, depending on the specific requirements of the task.

The Agent can be implemented using LLAmaindex’s RouterQueryEngine module which uses LLM, in this case GPT 3.5 to route the query to the tools provided. For this to be implemented, the individual RAG models with phi-3-mini and GPT-3.5 have to be packaged into tools and integrated with the RouterQueryEngine.

Walkthrough of individual model’s setup:

**RAG using phi-3-mini-instruct:**

* Skip this step if using GPT-4.
* To implement this RAG system locally, our options are limited to using small language models. Phi3-mini is one such model, part of Microsoft’s Phi-3 family, designed to be efficient and lightweight while maintaining high quality.
* We create a VectorIndex on the ingested data, integrating both the language model and the embedding model with the index.
* The LLM used is phi3:3.8b-mini-4k-instruct-q4\_K\_M, loaded via the Ollama package, while the embedding model, BAAI/bge-small-en-v1.5, is loaded using the HuggingFaceEmbeddings package from Langchain. The model utilizes 4-bit quantization, making it efficient for running on smaller devices.
* A query engine is generated using the index with the as\_query\_engine() function.
* Since creating the index is time-consuming, it can be stored locally for future access, which significantly speeds up query performance.
* The default prompts in the model can be customized to fit the specific use case, further refining the results.
* Finally, this setup is packaged into a tool that can be accessed by the Router Query Engine. [ insert code snippet for getting the LLM, embedding model, creating the vector index]

**RAG using 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.

In this case instead of a Vector Index , summary index is created/. In contrast with vector index summary indexes access the entire dataset for each query which provides answers with better context and is preferred for long form answers.

The llm used here is gpt-3.5-turbo and the embedding model used is text-embedding-ada-002 both of these can be loaded from the OpenAi module.

Similary to the previous model the summary index is integrated with the embedding model and LLm and a qyerty engine is created.

Finally we overwrite the default prompt to make the results suit our use case and store the index locally to be accessed in future uses.

[insert code snippet]

**RouterQueryEngine:**

Now that the tools are created. We setup the routerQueryengine and test it.

[router syntax]

**FastAPI and Uvicorn:**

This Rag system needs to be accessed by the automatuon code to parse queries and responses. For that we package it into an API and host it locally.

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]

Resulrs and Final thoughts:

-Sample Responses

-List of Jobs applied

-average time taken.