Instalily.AI SWE Case Study – Sarah Prakriti Peters

What does the Scraper.py file do?

The Parts Select website has been setup to showcase various appliances. If you click on refrigerators or dishwashers, it takes you to a page that lists all the popular parts associated with an appliance. The title of each part contains a link that takes you to a page that specially focusses on the part itself. Given that it is such a vast website, how do we access all this important information?

Solution: Scraper.py is setup to pull information from the most popular parts page. Some of the content includes manufacturer information, prices, reviews, and related links. All this information is stored in a CSV file, which can easily be read and analyzed using the Pandas package.

Code Map

1. Get fridge details/ Get Dishwasher details – This function pulls all important fields such as part numbers, titles, descriptions, and links for popular parts.
2. Pull Product Specs – For each product, we need to extract a list of replacements, symptoms that can be fixed, ratings and reviews.
3. Pull Q&As – For each product, we need to extract frequently asked questions and answers.
4. Store as CSV – Finally, write all this scraped data into a CSV file. CSV files are easy to handle and filter.

Advantages:

1. This file can be easily altered to include information on various appliances.
2. It automates the process of finding related links and stores the file locally.
3. It can be used to quickly assess if important information is being scraped or if we have too much noise in our dataset.

Disadvantages:

1. Extracting information is complicated because we must read the HTML tags. This varies for different webpages.

Is this scalable: For Parts select, yes – we can keep scraping important links and feeding it to the chatbot system. But if we were to do this for a different website, we would have to change the HTML tags.

What does the backend.py file do?

Code Map

1. **Collect important links** – the initial setup only contained links that were related to Popular dishwasher and refrigerator parts, returns, transactions, and purchases. These links are stored in a list called *imp_links*.
2. **Use scraped data** – Pass the extracted information using the scraper. These files were saved as CSV files, and the local paths are stored in a list called *file_paths*.
3. **Load data** – All documents must be loaded into a single object, and since we are dealing with CSV and web documents, I have used two loaders. The first is the CSV loader, which reads in file path and comma delimiter as arguments and is added to the *all_data* list. Then we use a Web loader, which takes *imp_links* as input.

4. **Split text** – The loaded information is too long, as we have provided an input of 2 csv files and 11 links. These documents must be broken down into smaller chunks that can be embedded. Each chunk has a size of 500 characters, and the overlap is 100 characters. We can think about this as a sliding window, where we try to include information in both the previous and next chunk, so no information is lost.
5. **Vector Database** – Set up the vector database such that it runs within the app. All documents that are used for context are split and fed into the database as chunks. Each chunk is embedded into a vector using OpenAI embeddings. This can be found in *vectorestore*.
6. **Retrieval Object** – The vector database is set up as a retriever, which is what will be used for context.
7. **Prompt template** – We need to define a prompt that limits the scope of the agent. We do not want the agent to answer any questions outside the scope of refrigerators or dishwashers, and so we can include a sentence like "*If there are any questions outside the scope of refrigerators, dishwashers, payments and returns, then please return an answer that says you cannot help with this query*". This is the prompt that will be passed to the large language model.
8. **LLM Setup** – Using ChatOpenAI, we can connect an LLM such as GPT-3 and fine tune it using hyperparameters such as temperature.
9. **RAG chain** – This is the object that will be used to produce an output every time the user submits a query. The rag chain contains the context (retriever object) and stores the user's query as a question. This is then connected to the prompt that is passed to the LLM, which generates the output which will be printed after calling the invoke function. This entire setup needs to be done once, after which we can use the invoke method for every query. Hence, this setup is coded as a function that is called once.

Why use a Vector Database and not a vector library?

One of the main advantages of using vector databases over libraries is the ability to store and update data. Since they allow for CRUD operations (create, read, update, and delete), they overcome issues in vector libraries like immutable indexing. Weaviate is an open-source vector database that converts the query to a vector and finds objects in the database that have vectors like the vector query. The default vectorizer (which determines how data vectors are created) is text2vec. Weaviate, by default, uses cosine similarity to extract matching vectors.

Advantages:

1. This code can be altered to add more links or CSV files, which means we can always feed in more context.
2. The code to set up the RAG chain only runs once, which makes this code run faster.

Disadvantage:

1. Each CSV file is uploaded locally, which could be a memory issue if we have too many CSV files.

Is this scalable: Yes.

What does the backend_with_improvements.py file do?

The general code map follows the same structure as backend.py. This file is used for experimenting with each component –

1. **Prompt Engineering** – The easiest way to limit the chatbot from answering questions outside the scope is by telling the LLM to not answer questions outside the scope. This is what we call a prompt, and is fed to the LLM, like how we interact with ChatGPT. Personally, I felt that developing prompts is more creative than scientific. The LLM may not interpret the prompt that way you intended, and it requires a lot of tweaking.
2. **Increasing Context** – In the original backend, only 11 links and 2 CSV files were fed in as input. This provides limited context, and the system may not be able to answer a lot of queries. Also, using two types of inputs could be harder to scale. Using the scraper file, we pull additional links, and store it in the *imp_links* list. This creates an input of 99 documents.
3. **Chunking** - Initially, I used a CharacterTextSplitter which would split the documents based on a character count. In some cases, the chunk sizes would surpass the character count and I would be left with larger chunks. Also, having too large chunk sizes resulted in poorer outputs. An alternative was to recursively chunk, for which I used the RecursiveTextSplitter.
4. **Embeddings** – Initially, I used OpenAI Embeddings. This is based on the OpenAI API, and only requires an API key. However, I was unsure if these embeddings captured information well, which is why I decided to experiment with other options such as HuggingFace embeddings. HuggingFace embeddings map sentences & paragraphs to a 384-dimensional dense vector space and can be used for semantic search.
5. **LLMs** – GPT 3.5 Turbo is a powerful LLM, but it still does not provide useful answers all the time. One option to improve the results were by using alternate GPTs, Claude, et cetera.
    a. **Hyperparameters in the LLM** –
        i. **Temperature** – Similar to a reproducibility score, lower temperature values will make the output more deterministic.
        ii. **Max Tokens** – The responses were sometimes longer than expected, which resulted in errors. To limit this, I set max tokens to 1000.

LangChain

LangChain provides a set of tools to set up the Retrieval-Augmented Generation (RAG) system. Some of the key modules are as follows –

1. **Options for LLMs –** LangChain allows us to implement multiple LLMs such as GPT and Claude using *langchain.chat_models*. Each model just requires an API key to be set up.
2. **Prompt Templates –** *LangChain.prompts* can be used to create a simple template that can be attached to the RAG setup. This prompt is passed to the LLM to limit answers to unrelated queries.
3. **Chain –** LangChain offers an easy setup to link multiple steps such as data processing and LLM calls.
4. **Text Splitting -** *langchain.text_splitter* is used to create chunks, that are embedded into the vector database.
5. **Document Loaders –** *langchain_community.document_loaders* contain options to load different forms of data such as CSV, PDFs, web links, et cetera.
    a. **LangChain Web Loader –**
    The Web Base loader uses BeautifulSoup to extract data from the HTML code. My approach inputs relevant product links, which are loaded through the web base loader. This should extract all the text from the relevant links. Once the data is loaded, it is stored in *all_data* which is fed to the text splitter to create chunks.

Post the meeting, I checked my web-loaded data, and noticed gaps in the scraped text. This would explain the issues in my customer chatbot. However, my initial setup included the scraper file, which I manually set up to scrape a web page. This file can be used to replace the web-based loader.