Ocado Web Scraper

We built this commercial scraper to scrape all product and recipe data from Ocado.com, an online supermarket. The website holds thousands of products and hundreds of recipes and we scraped all of this data (+75K product pages).

For each product, we obtained almost all the information contained within the product page. Details like price, description, ratings, images and whether the product was out of stock.

This information could be used to compare products at other supermarkets or to perform future price predictions.

Implementation

The way the scraper gathers information is as follows:

1. Get the URLs of all the top levels categories (those found in ‘Browse’ on the homepage) on the ocado website and how many products there are in each category.
   Iteratively for each category:
2. displays all products in the category and scrapes and saves all product urls in that category
3. Using the product urls found in the previous step, scrapes each product page for information and images.
4. Saves the category data locally from the previous step to a nested dictionary (see Json structure below). Note image links are also saved and images downloaded locally if specified.
5. Uploads the raw data json and downloaded images to a data lake (AWS S3 bucket).
6. Processes the json raw data in pandas to create several dataframes to enable easy creation of normalised SQL tables.
7. Exports the normalised sql tables to store in an AWS Postgres RDS database.

Json Structure

```
{
Bakery : {
            sku1 : {name: ,....},
            sku2 : {            },
            ..........................
            ..................
           },
Food Cupboard : {
                  sku1 : {name: ,....},
                  sku2 : {            },
                  ..........................
                 ..................
                  },
..............................
..............................
}
```

Multithreading

We developed the project using multithreading in order to improve the performance and scrape thousands of products as efficiently as possible.
Ovi to write ...

Docker and EC2

In order to eliminate local environment dependency and make it easier to run the scraper on virtual machines or the cloud we containerised the scraper using Docker (see below for image details on Docker Hub).

We used selenium in headless mode to improve speed and performance and this was also essential for Docker as a container cannot open browsers in graphic mode.

We ran the docker container on the cloud using an AWS EC2 instance.

Monitoring in Prometheus

We monitored the performance of the scraper using Prometheus....

Data post processing

3 classes created to:

- Upload the raw data and downloaded images to an AWS S3 Data Lake by entering the bucker details
- Process the raw data json in pandas to create several dataframes that can be exported directly into SQL
- Export the SQL tables created from the dataframes in the previous step to an AWS RDS Postgres database by entering the the database endpoint and password.

Installation/Running the scraper

Package available on PYPI: <insert link>

Docker image: <insert link to the image on dockerhub>

Running locally:

- OcadoScraper.py contains the scraper class needed to launch a new scraper and scrape all the products on the website.
- It is also possible to scrape a subset of categories.
- There are several public functions to provide information:
    - the categories and products available to scrape on the Ocado website.

    - about the status of the scrape

Example usage:

- create a new instance of the scraper class: `ocado = OcadoScraper(scrape_categories=True, )`
- Check which categories are available to scrape on the ocado website: `ocado.categories_available_to_scrape()`
- Scrape all products on the ocado website: `ocado.scrape_products(download_images=False)`
- OR scrape a subset of categories: `ocado.scrape_products(['Bakery', 'Frozen Food'])`

Information functions:

Public function to get information about the data scraped before, during and after the scrape

- Show status information about the scrape - eg which categories have been scraped already and number of products scraped, also which categories are left to scrape etc : `ocado.current_status_info()`
- Scrape one product to view sample data collected and download it's images

```
url='https://www.ocado.com/products/hovis-best-of-both-medium-sliced-226160'
data = ocado.scrape_product(url, True)
print(data)
```

- Display a list of categories without saved product data:

  `ocado.get_categories_without_saved_product_data()`

- Display the number of products in each category:

  `ocado.number_of_products_in_categories()` Note this may be from saved data so if this is the case and we want to get the latest data run `categories_available_to_scrape(from_file=False)`

- Download images for the categories specified after the scrape has been run. Default parameter is all categories: `ocado.download_images(['Frozen Food', 'Bakery' ])` (Note images can also be downloaded during the scrape by setting `download_images=False` in `scrape_products()`)
- There are also public functions to delete the saved product data file, downloaded images and to delete the product data for a specific category:
    - `delete_saved_product_data()`
    - `delete_saved_category_url_data`
    - `delete_saved_product_data_for_category("Bakery"):`
    - `delete_downloaded_images()`

Testing

Dilan to write ...

Development process

We initially started planning by looking at what the requirements would be from a user perspective and we decided we needed the ability to scrape by category.
We looked at the structure of the site to see what was feasible and started by scraping the data on a few product pages and getting all the products in a category to test the design before expanding the scraping the whole site.

To ensure the data would be available to various end users the raw data was stored in a S3 data lake and then processed in pandas to make it simple to export the data to five sql tables (normalised to structure the list entries in the correct way) in an AWS Postgres database.

We chose Postgres because .. TODO
We chose AWS because ..TODO

Unexpected challenges

We did have some unexpected challenges throughout the project:

- TO DO

Future developments:

- Scrape by category url
- Add a Flask or Tkinter UI
- Public API for the scraped recipes
- Link the recipe and product data together eg for a particular product show all recipes containing that product etc