Austin Strain
8/3/2020
1487409
University of Houston

Research Report

GitHub:https://github.com/mudabbirk/Summer-Research/tree/master/Austin

Introduction

Over the summer I have been collecting data on product details, more specifically data that is used to identify products across different platforms. The goal is to understand how different Data elements are exchanged between e-commerce players. We can find answers by studying different business standards such as product ID, shipping labels, tracking number, barcode, and etc... For shipping, we can look at standards like ISO, GS1, WCO, and UNCEFACT. The result of the research should be a map of how data flows among e-commerce players.

I looked at several different approaches to this problem, I looked at shipping API's from USPS, UPS, FedEx, etc…, UPC API's, eCommerce platform API's from Wish, eBay, Amazon, etc.. , and web scraping tools. I found that the web scraping tools were the easiest to work with and provided useful data.

API's

API's, or application programming interface, are basically HTTP requests to a server that return data. Using an API you typically need to specify a user token/username, a password, the URL, and service you'd like to use. A username and password is generated by signing up at the respective API's website. Most, if not all API's, have instructions or a user manual that include code samples. You can use programs such as Postman that help make these API requests easy.

For Postman, all that is required is the type of HTTP request (most likely going to be a GET) and the URL. Postman can return all types of data formats including HTML, LXML, JSON, and more. Postman is good for testing out how API services are used and how they respond. It is not good for making multiple requests at a time.

UPC: https://github.com/mudabbirk/Summer-Research/blob/master/Austin/UPCitem_API.py/UPCitem_API.py

UPC, or Universal Product Code, is the standard ID for products used around the world. Some European countries may use a EAN, or European Article Number, if that is the case the EAN will be the UPC. In the GitHub link, you will find a program that makes a request to https://www.upcitemdb.com/. The python program “UPCitem_API.py” takes a list of UPC’s from the text file "UPC_EAN_list.txt" and makes an HTTP GET request to the upcitemdb website and return the products data in a JSON format. Then the program will parse the JSON and output it into an XLSX file.

The data received will include the EAN, title, UPC, GTIN, EILID, description, brand, model, color, size, dimension, weight, category, currency, price, lowest_recorded_price, highest_recorded_price, images, merchant, user_data, and ASIN. Not all of these attribute fields are required, some fields may be blank.

This would be the perfect solution for collecting product information if it wasn’t for two things. The first being able to get a large list of UPC’s to use. The second being that this UPC database, like many others, limits the number of HTTP requests that can be made in a day for free.

eCommerce

Many different eCommerce websites like Amazon, eBay, or Wish have their own API services as well. These API services are mostly used for sellers who want to automate their listings. However, some are made for buyers and will return product information. Amazon has an API marketplace that returns mass amounts of products, however it costs around $30 a month.

I did not develop any programs that use these API’s. The reason for this is that the data that is returned is limited and is not geared for data collection. These API’s are also often restricted to users who are sellers or buyers.

Shipping

Just about every postal service has an API’s. I did not develop any programs for this API for similar reasons as the eCommerce API services. Most of the API’s are geared towards sellers or buyers that are trying to automate their business. These API’s would have been useful to see how different standards track data. For example, what are the different barcodes used and when are they used?

These API’s services are limited to those who are just trying to collect data as well. Many of them require a tracking number or a specific ID. This can be a roadblock if you do not have access to this information in mass quantity.

Web Scrapers

I have found that web scrapers are the most useful and easiest to use. Web scrapers work by making an HTTP request to a website, retrieving the HTML and parsing it for specific details.

There are several different Python web scraping technologies. The one that I ended up using is called Beautiful Soup. I attempted to use Parsehub and Scrapy in the past. Parsehub does not require any coding and relies on a GUI and is filled with bugs. Scrapy seemed overly complicated and had too many unnecessary moving parts.

Beautiful Soup works by taking an HTTP request and an HTML format and creates a “soup” object. Soup objects contain the HTML from a page and has methods that allow you to find specific HTML tags. For example:

`liTag = soup.find(‘li’ , attrs = {‘class’ : ‘product weight’})`

This will find a “li” HTML tag where the class is product weight. If there is no tag with those parameters the “find” method will return ‘None’ or throw an exception.

File format and framework

The file format for most of the web scrapers are similar if not the same. The file paths may need to be changed. For example

`txt = open("keywords.txt","r+")`

May need to be changed to

`txt = open("amazonscraper/keywords.txt","r+")`

There is a collect_links.py, a .csv file, inputKeywords.txt, main.py, outputKeywords.txt, keywords.txt, README.md, scrape_page.py, proxies.txt, and proxy_authenticator.

collect_links.py will take a search term and sometimes the number of pages of results that you’d like to search and return a list of products URL’s.

The .csv file is where all the saved data from each product is recorded.

The inputKeywords.txt only used by the Craigslist scraper is a list of search terms that the program iterates through, rather than taking in search terms from the command prompt.

main.py is the python file that you will want to execute. It will run all of the needed functions. main.py will create the .csv file and define the headers. It will prompt the user if they would wish to append to the data file or overwrite it. It will call the functions in collect_links.py and track all of the products URL. Then for each product URL, it will make function calls to functions in scrape_page.py. For each product, main.py will gather data attributes such as the title by doing:

`title = get_title(soup)`

get_title is a function in scrape_page.py that will use the “find” method to collect the corresponding attribute.

outputKeywords.txt only used by the Craigslist scraper contains a list of search terms used and the number of products that were collected using the corresponding search term.

Keywords.txt has the same purpose as the outputKeywords.txt but is only used by the Amazon and eBay web scraper.

The README.md will give basic instructions, a description of the program, and a list of required libraries.

The scrape_page.py is where the magic happens. This file will 'scrape' each product page for specific data attributes. If the function call to get an attribute returns nothing an exception is thrown and the attribute is set to 'N/A' (for Amazon) or ' '(for Craigslist and eBay).

proxies.txt is used by the Amazon web scraper and contains a list of proxy IP addresses that can be used for HTTP requests.

proxy_authenticator.py is only used by the Amazon web scraper, grabs random IP addresses from proxies.txt, and tests whether or not it is valid and returns a legitimate and accurate HTML. If it does not, it grabs another random IP address and retries. If it does work then it creates a soup object to return to main.py.

Amazon's firewall

Amazon tries very hard to not allow web scrapers. This is most likely because they want people to pay for the marketplace API. If Amazon detects that the HTTP request is made by a web scraper it will redirect you to a different page, or it will simply return no HTML to the soup object. I have worked around this issue by rotating IP addresses, making random waits between HTTP requests, and defining the requests header. Because of this, the Amazon web scraper takes nearly 4 times as long as the others.

I use a list of free proxy IP addresses fromhttps://proxyscrape.com/free-proxy-list. I set the country to a country that uses English so the HTML returns in English. I have had the best luck when anonymity is set to transparent, and SSL is set to yes. The proxy IP addresses are saved in the proxies.txt. Every new request that is used to collect links uses a new IP address. Data attributes are collected on each product URL page until it hits a firewall then a new IP address is used.

- Amazon's different page layouts

This may be another attempt by Amazon to avoid being scraped or an attempt at making pages easier for the user to read. Depending on the product, often depending on the category, the attributes of each product are displayed differently. For example, some attributes are formatted in a table and some are in bullet points. If you look at the functions in scrape_page.py you will see how I attempted to tackle this problem by using multiple try and except commands. With attributes being displayed differently, the HTML will be different, therefore the same "find" method will not be applicable to different products.

Craigslist different URL's for different cities

Craigslist uses different URL's for different cities. For example, Houston uses:
https://houston.craigslist.org/
Meanwhile Austin uses
https://austin.craigslist.org/

This creates a bit of a complication. Using a search term will only return items that are in the respective city, therefore the main.py prompts the users to enter the city that the user would like to collect data from. A list of cities on craigslist can be found at https://geo.craigslist.org/iso/us. This also means that a search for an item won't return many results compared to the other scrapers. I combatted this by having the Craigslist web scraper make searches based on words in the inputKeywords.txt file. So every time it executes, it collects data on all of the search terms in the file for the city that is entered.

Next steps

- Improve the speed and accuracy of the Amazon web scraper.
  - Some other web scraping technologies use a 'middle wear' that can automate using different IP addresses that may help with avoiding the firewall much faster. The accuracy could be improved with more testing and improvements. As mentioned, Amazon uses different HTML tags to list product attributes. There are most likely different listing styles that I have not accounted for yet.
- Combine the data into a database
  - This will allow the data to be more accessible and easier to analyze.
- Collect data on shipping data
  - There are many different roles involved with eCommerce and shipping is a huge component of it. Collecting data and analyzing how shipping standards use data will allow us to get a better map.