A Cloud-based System for Scraping Data From Amazon Product Reviews at Scale

Ryan Woodall
Jrw5074@uncw.edu

Douglas Kline
klined@uncw.edu
Information Systems

Ron Vetter
vetterr@uncw.edu
Computer Science

Minoo Modaresnezhad
modarsm@uncw.edu
Information Systems

University of North Carolina Wilmington
Wilmington, NC 28403

Abstract

Amazon product reviews can provide a rich source of data for natural language processing research. To support a related research project, we built a custom cloud-based system for obtaining Amazon product reviews. A third party cloud-based scraping service automatically retrieved scraping jobs, then notified Azure Data Factory through an Azure Function. Raw scraping data was then transferred in batches to Azure Data Lake Storage, then custom SQL transformed the data for convenient query from an Azure SQL database. The system was used to obtain 17,962 product reviews and produce data sets in several formats. This paper fully describes the system, and offers lessons learned from the experience.

Keywords: Data Pipeline, Cloud, Amazon Reviews, Big Data, Azure.

1. Introduction

Modern companies struggle with big data collection and processing, and it has become best practice to accomplish this with data pipelines in the cloud. These systems need to be maintainable, adaptable, repeatable, and scalable. To explore modern cloud-based data-oriented system development, we created a system to gather large numbers of Amazon product reviews.

Amazon reviews are important for researchers exploring natural language processing. Each product has a distinctive dictionary, i.e., the words used in reviews change for each product category and each product. Reviews offer the ability for researchers to evaluate methods across challenging situations such as dictionary, data quantity, data quality, themes, sentiment, etc.

Several Amazon Review data sets are freely available online. The data set used for McAuley, et al. (2015) and He & McAuley (2016) was primarily gathered for research relating the review text to product images. This data set is quite large covering many categories and products. However, it is somewhat dated (May 1996 - July 2014), and does not include some of the review quality features implemented by Amazon, such as verified purchases. A subsequent dataset was collected by Ni, et al. (2019) providing more recent reviews (through 2018), but still lacks the newer review quality indicators, as well as details about the reviewer that can be used to filter fake reviews.

Amazon (2019) offers their own large review data set, which includes whether the review was tied to a verified purchase. This still leaves out many data items that can be used to evaluate the quality and reliability of a review such as reviewer name, reviewer rating, reviewer social media names, etc.

A number of for-pay services exist providing functionality similar to that created in this project. However, the actual data pipeline is opaque with no visibility of the transformations occurring to the data. Amazon itself has recently begun offering a for-pay API for accessing their data. However, API access tokens are given only to developers from vetted companies, and we did not explore this avenue.

Amazon reviews can be used for a number of natural language processing (NLP) research purposes such as sentiment analysis, bot detection, theme analysis, summarization, and recommender systems. Furthermore, reviews are the primary method for consumers to evaluate products for purchases.

For a related research project (Gokce, et al 2021) we decided to collect a custom data set that would be more current and include the missing items identified above. We used this opportunity to review modern methods for large scale collection of data in the cloud. Rather than filtering the existing massive data sets for the data needed, we would create smaller targeted sets created for our tasks. Another concern we had was that the actual processing steps performed on the raw reviews is unclear, and perhaps not repeatable. Ni, et al. (2019) offers their data in several forms with different levels of "aggressive" removal of reviews for various reasons. We took this opportunity to curate our own data set in a fully auditable, repeatable manner, where every data modification was explicitly described by code.

To fully explore modern data pipeline methods, we established the following goals:

- Cloud-based – the system should entirely reside in the cloud
- Automated – the system should orchestrate tasks without in-progress intervention
- Scalable – the system should be scalable
- Formats – the system should offer the data set(s) in several formats

2. Technologies Used

In this section we describe the technologies that we used and why they were chosen. Many vendors offer cloud-based services. To limit the overwhelming options and to align with our existing technology skillsets, we chose to use Azure cloud services as much as possible. Generally equivalent services exist on most cloud provider platforms.

We used a third party cloud-based web scraping service called WebScraper (2020) because of its convenient low-code nature and our prior experience with its desktop-product. Scrape job definitions can be authored in the Chrome web browser, and exported in JSON format. Scrape jobs can traverse paginated web pages, drill down and up through pages, and gather related data entities such as product, review, reviewer, etc. Bot-detection-avoidance features are included such as pauses between page requests, and the use of multiple IP addresses for requests. In the cloud-based service, a full API is provided for programmatic definition of jobs.

Azure Data Lake Storage Gen2 was used to store raw files, and also as the storage area for the Azure SQL instance. This service offers a hierarchical name space, high scalability, metered fees, and can support map-reduce style operations.

Azure Data Factory (ADF) integrates seamlessly with Data Lake Storage, and can be used to orchestrate data workflows from various locations and services in the cloud. Our initial intent was to use ADF for the bulk of the data operations, but ultimately, it was mainly used to move data from one location to another.

An Azure SQL relational database instance was used to deliver final data in an easily query-able format. In addition, relational models offer a quite compact representation of the data, reducing storage costs, and improving performance.

Azure Functions were used to trigger events across distributed components of the system. With Azure Functions, a secure HTTPS endpoint could be called with proper credentials, triggering events in a remote system and/or passing data between systems.

We planned to use Azure Key Vault to manage the secrets needed for secure communication between distributed system components. Secrets we planned keep in the Azure Key Vault included connection strings, credentials (username/password), and API tokens. Ultimately, Azure Key Vault proved unwieldy and required a full Azure Active Directory Doman and high level credentials. For expedience without sacrificing security, secrets were stored in each linked service defined in Azure Data Factory.

The many cloud services were declaratively defined in Azure Resource Manager (ARM) templates. ARM templates define the desired end state of a collection of services, and manage the connections and security concerns among the services. ARM templates are text-based and declarative. With a system's ARM template, and complete replica of a complex distributed set of cloud services can be perfectly replicated. Furthermore, the entire system can be versioned in source control.

Even with the cloud-based services used, there were places where programming code was required. In particular Transact SQL, Microsofts procedural scripting language, was used to transform data from a staging table to relational tables in the database. Python code was used in the Azure function.

3. System Description

In this section, we describe the system as it was ultimately built. The System Overview diagram in the appendix gives a high-level view of the major components. The Webscraper component was the only non-Azure part. All other components were housed in a single Azure Resource Group from which an ARM template could be generated, completely defining all components and their interactions.

The system operated as a set of workflows:

- Product Loader
- Data Scrape
- Data Retrieval
- Process Reviews
- Flat-file creation

The Product Loader is in charge of looking for new product review scraping jobs and starting them. New scrape jobs are entered (by humans or otherwise) in the ProductURLs.csv file in Azure Storage. A scrape job is defined by the url of a product on Amazon. The same scrape definition is used on all products. The Product Loader is implemented as a periodic (every 15 minutes) azure data factory pipeline. For every URL in the file, a POST is made to the WebScraper service API using secure credentials. Status information is stored in various files in Azure Storage.

The Data Scrape is performed over time by the WebScraper third party service. When it is complete, the WebScraper service POSTs to an Azure Function exposed as an HTTP endpoint. This message does not transfer the data, but merely indicates that the scrape job has been completed. The Azure Function is implemented in Python, and simply records completion of the task in the database.

The Data Retrieval component is implemented as a periodic azure data factory pipeline, similar to the Product Loader. For each completed scrape job, it makes an API call to the WebScraper API for the resulting data and metadata, and moves the data to Azure Storage. A single product produces single scrape job, which produces a single set of reviews, in a single file. For each product, the Process Reviews component in triggered.

The Process Reviews component moves a file of reviews into a staging table in the Azure SQL database. The data is inserted into the staging table via an efficient bulk-load operation with no integrity checking. The data is unmodified and enters the table with all character-based data types. A stored procedure written in Transact-SQL transforms the data and inserts it into the Product, Review, and Reviewer relational tables. SQL is used to perform efficient set-at-a-time operations with minimal data transformations. Notably, the star-rating is transformed from prose ("one star out of five") to an integer data type.

Finally, the Flat-file creation component executes a SQL query to produce a CSV flat-file, as well as a hierarchical JSON file. This is for convenience, and represents the formats which would be publicly published for general analytics use. Because the data is in a relational schema with a database, any subset of the data can be readily produced in any format using straightforward SQL.

4. Results & Discussion

To test the system reviews for 15 products across 9 subcategories in the Audio Books & Originals category were collected. The subcategories were:

- Bios & Memoires
- Self-Development
- Literature & Fiction
- Business & Careers
- Science Fiction & Fantasy
- Teen & Young Adult
- Health & Wellness
- Computers & Technology
- Kids

In total 17,962 reviews were collected and processed through the pipeline, producing data in CSV and JSON formats, as well as recording all entity instances in the relational database. Each form of the data was retained, providing a full audit-trail of all changes to the data.

The Webscraper service made requests in three parallel threads emanating from separate IP addresses, with adjustable delays between page requests. The number of reviews per product ranged from 293 to 2690, and scrape times per product ranged from 1.5 to 9.75 hours with a combined scraping time of 67.5 hours. The number of records scraped per hour average approximately 266.

In general, we accomplished the goals set out at the beginning of the project. The system can produce large amounts of Amazon reviews in a variety of formats in an automated manner.

By creating the system, we learned much and will relate some opinions and advice for developers implementing similar systems with these technologies.

Azure Data Factory & Azure Functions
It became clear that Azure Data Factory and Azure Functions were not fully mature products, and in cases changing during the development of this system. First, overnight changes in features broke the system multiple times during development, requiring rewrites of code. Second, integrations with products and languages were not complete. For example, when using Python to write Azure Functions, there was no direct access to Azure SQL, while there was direct access to CosmosDB.

We found Azure Data Factory jobs difficult and costly to debug, as well as expensive and inefficient. Processing appears to be row-by-agonizing-row, which was unnecessary in our situation. Ultimately, parts that we intended to write in Azure Data Factory were implemented as set-at-a-time operations in SQL. We found this to be more transparent, easier to write and debug, and ultimately much faster than the ADF pipelines.

Third Party Cost
We chose to use the cloud-based WebScraper service for its low-code approach and our familiarity with the desktop product. However, this service was the majority of the cost for producing reviews. The cost structure does not scale to the level we want. This is not critical of the service; it worked well, as advertised, and required minimal effort.

To continue collecting reviews at scale with reasonable costs, we anticipate custom-writing the review collection component. There are a number of libraries available, including python libraries Beautiful Soup (2021) and Scrapy (2021). We would not have to write a general purpose full-featured cloud-based web scraping tool with all features of WebScraper, but could write code specific to collecting Amazon Reviews. It appears feasible to implement a custom scraper in one or more virtual machines or containers, or even in an Azure Function. A less elegant approach would involve running the desktop browser-based free version of WebScraper in virtual desktops and collecting results as they are produced.

Cloud-based Considerations
The cloud-based distributed architecture comes with benefits and challenges. A distributed architecture necessitates decoupling and explicit interfaces between components, which generally produces a very clear transparent system. However, secure communication becomes onerous compared to a monolithic application. In systems with more components and services, the need for secrets management would be required through a product like Azure Key Vault.
ARM templates provide the ability to persist the exact definition of the web of services a complex system in a text-based, version-able form. This opens the door to team development and change management that didn’t exist pre-cloud. However, each service we used was quite complex in its own right, each imposing its own learning curve. Ultimately, the logic of the entire system was spread across many places. In this cloud-based environment, it is very important to have clear roles for each component, and explicit interface contracts to manage interactions.

5. ACKNOWLEDGEMENTS

We wish to recognize the support of the Congdon School of Supply Chain, Business Analytics, and Information Systems at the University of North Carolina Wilmington.

6. REFERENCES

Woodall, R., Kline, D, Vetter, R., Modaresnezhad, M. (2021) A Data Pipeline for Amazon Review Collection and Preparation. Annals of the Master of Science in Computer Science and Information Systems at UNC Wilmington, 15(1) paper 3. http://csbapp.uncw.edu/data/mscsis/full.aspx.

Webscraper Documentation. (2020). Retrieved September 5, 2020, from https://webscraper.io/documentation

Amazon Web Services Open Data, 2019. Multilingual Amazon Reviews Corpus https://registry.opendata.aws/amazon-reviews.

Gokce, Y., Kline, D, Vetter, R., Cummings, J. (2021) Automated Text Reduction: Comparison of Reduced Reading List Creation Methods. Annals of the Master of Science in Computer Science and Information Systems at UNC Wilmington, 15(1) paper 2. http://csbapp.uncw.edu/data/mscsis/full.aspx.

He, R., & McAuley, J. (2016, April). Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In proceedings of the 25th international conference on world wide web (pp. 507-517).

McAuley, J., Targett, C., Shi, Q., & Van Den Hengel, A. (2015, August). Image-based recommendations on styles and substitutes. In Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval (pp. 43-52).

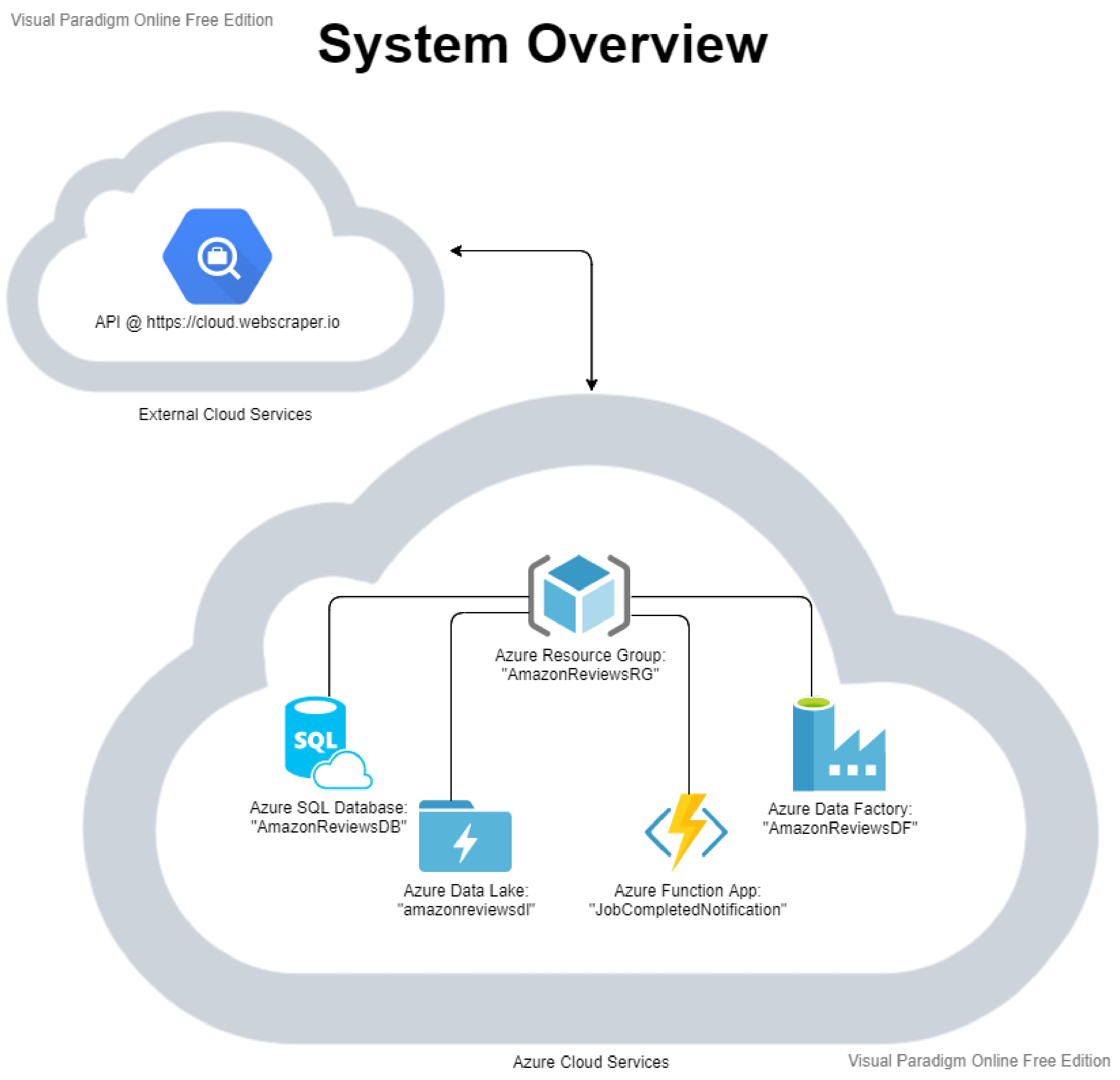
Ni, J., Li, J., & McAuley, J. (2019, November). Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP) (pp. 188-197).

Beautiful Soup (2021) https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

Scrapy (2021) https://scrapy.org/.

.

Appendices


Visual Paradigm Online Free Edition
System Overview
API @ https://cloud.webscraper.io
External Cloud Services
Azure Resource Group: "AmazonReviewsRG"
Azure SQL Database: "AmazonReviewsDB"
Azure Data Lake: "amazonreviewsdl"
Azure Function App: "JobCompletedNotification"
Azure Data Factory: "AmazonReviewsDF"
Azure Cloud Services
Visual Paradigm Online Free Edition