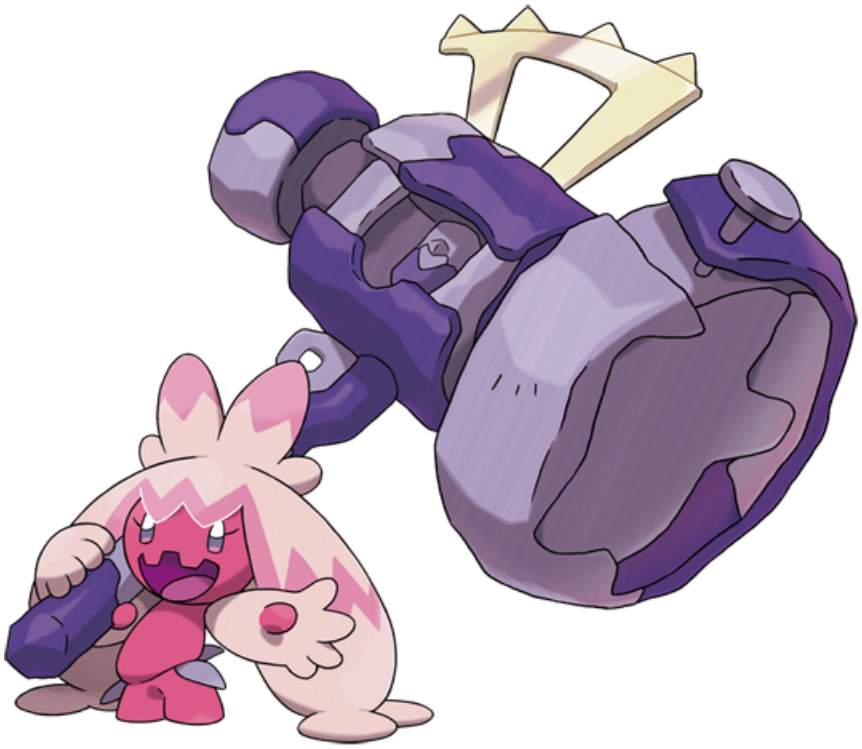
GROUP NAME

Tink A Ton



ASSIGNMENT 3

Amazon Reviews Analysis

GROUP MEMBERS

DMITRI carrot2803

LORENZO Enzo-TTD

FELICIA meiying13

Introduction

Online customer reviews are a critical source of information for both consumers and businesses, shaping purchasing behavior and informing product development. This assignment leverages the **McAuley-Lab/Amazon-Reviews-2023** dataset - approximately 200GB of Amazon product reviews and metadata across 34 categories.

Our work is structured into six key components: data acquisition, data cleaning, exploratory data analysis (EDA), sentiment prediction, recommendation modeling, and clustering. To process this large dataset, we leveraged **Polars** due to its performance in low-resource environments, scalability, and our prior expertise.

In contrast to an alternative like **PySpark**, which demonstrates suboptimal out-of-the-box performance, particularly on weaker compute, and requires additional DevOps effort to scale. Even then, **PySpark's** performance at scale is often inferior to **Polars'**, reinforcing our decision to choose **Polars** for this project.

Other tools we relied on included **Plotly**, which enabled us to save our visualizations and efficiently render large datasets while also enhancing data interpretability. We also made extensive use of the `sklearnex` library to accelerate computations with scikit-learn, taking advantage of the fact that all our CPUs were Intel-based.

By making these thoughtful decisions, we were able to utilize the **full dataset** for all tasks.

Data Acquisition (Task 1)

To acquire the dataset, we implemented a custom ingestion loop integrated with our medallion architecture (see Data Cleaning). For each of the 34 categories, the loop retrieves both reviews and metadata, writing each split as a raw Parquet file in the Bronze layer using a custom ingestion function, `download_data(cat, type, ram)` (See Figure 1).

This function also includes an optional RAM flag to keep datasets in memory on high-RAM machines. After each write, the data objects are explicitly deleted, and the **garbage collector** is triggered to free up memory, thereby accelerating **throughput**. These performance decisions were made as RAM was a limiting factor.

All 34 categories were successfully ingested and persisted to disk (See Figure 2).

Aquisition Pipeline

```
from amazon.utils import (
    create_directories,
    download_data,
)
from amazon.constants import *

for category in ALL_CATEGORIES:
    print(f"Downloading {category}...\n")
    download_data(category, REVIEW)
    download_data(category, META)
```

Aquisition Pipeline

```
import gc
import os
from datasets import load_dataset, Dataset

def download_data(category: str, type: str, ram: bool = False) -> None:
    path: str = f"data/raw/{type}/{category}.parquet"
    if os.path.exists(path):
        print(f"File already exists: {path}. Skipping.")
        return

    print(f"Downloading {category} ({type})...")
    data_set: Dataset = load_amazon_dataset(category, type, ram)
    data_set.to_parquet(path)
    del data_set
    gc.collect()
```

Figure 1: Data Acquisition Pipeline

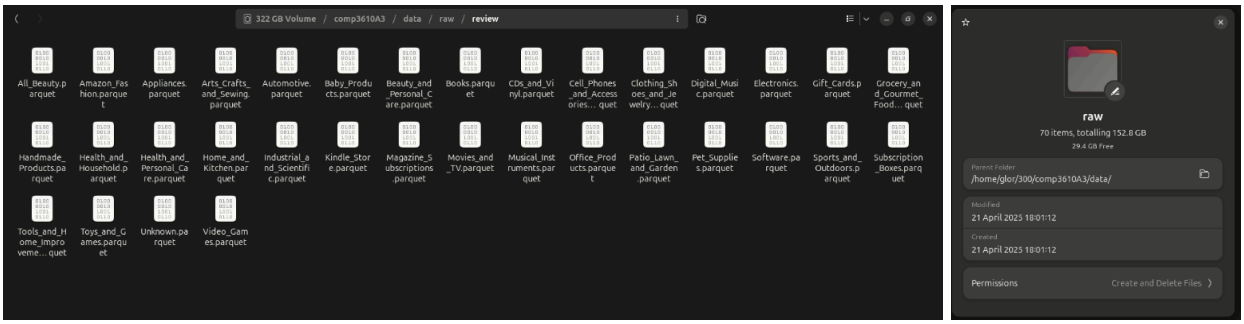


Figure 2: Successful Acquisition of All Categories

See More

Data Cleaning & Preprocessing (Task 2)

Due to the assignment's explicit data cleaning guidelines, our implementation prioritized architectural design over subjective decision-making. Review data was merged with metadata on `parent_asin`, with invalid star ratings and empty texts removed, and missing brands set to "Unknown." Duplicates, based on user ID, product ID, and review text were dropped. Finally, token count and review year were derived from the review text and timestamp. As instructed.

The architecture chosen was the **medallion architecture**, refining this data across three layers: **Bronze** for raw data (Parquet, JSONL, HuggingFace Dataset, etc.), **Silver** for clean and formatted intermediate data, and **Gold** for processed, query-ready data. The separation between the Bronze and Silver layers allows the acquisition and processing phases to be decoupled in case of breakage, and also helps reduce RAM consumption over short periods. The separation of **Silver** and **Gold** also enables a clear split between usage of the combined data and processing of the intermediate data.

While the main advantage of the **medallion architecture** is reduced local compute (pro), it comes with increased storage requirements (con). This can be mitigated through Parquet compression, however, compression reintroduces the need for compute and RAM overheads.

For the cleaning process, we used **Polars'** lazy API, leveraging `scan_parquet` for lazy loading to create execution graphs without immediately loading data into memory. All cleaning operations, including removing invalid entries, deduplication, and joins were executed in parallel using **Polars'** C++ backend. The resulting clean data was then written to disk using `sink_parquet`, with **Polars'** streaming engine ensuring that data was never fully materialized in RAM, as we were RAM-bounded, not time-bounded.

We were able to merge all categories and produced a unified dataset containing 502,984,947 rows, which was used in full for downstream tasks.

Dataset Shape: (502_984_947, 27)

```
merged.collect(engine="streaming").shape
✓ 0.0s                                                    Python
(502984947, 27)
```

Exploratory Data Analysis (Task 3)

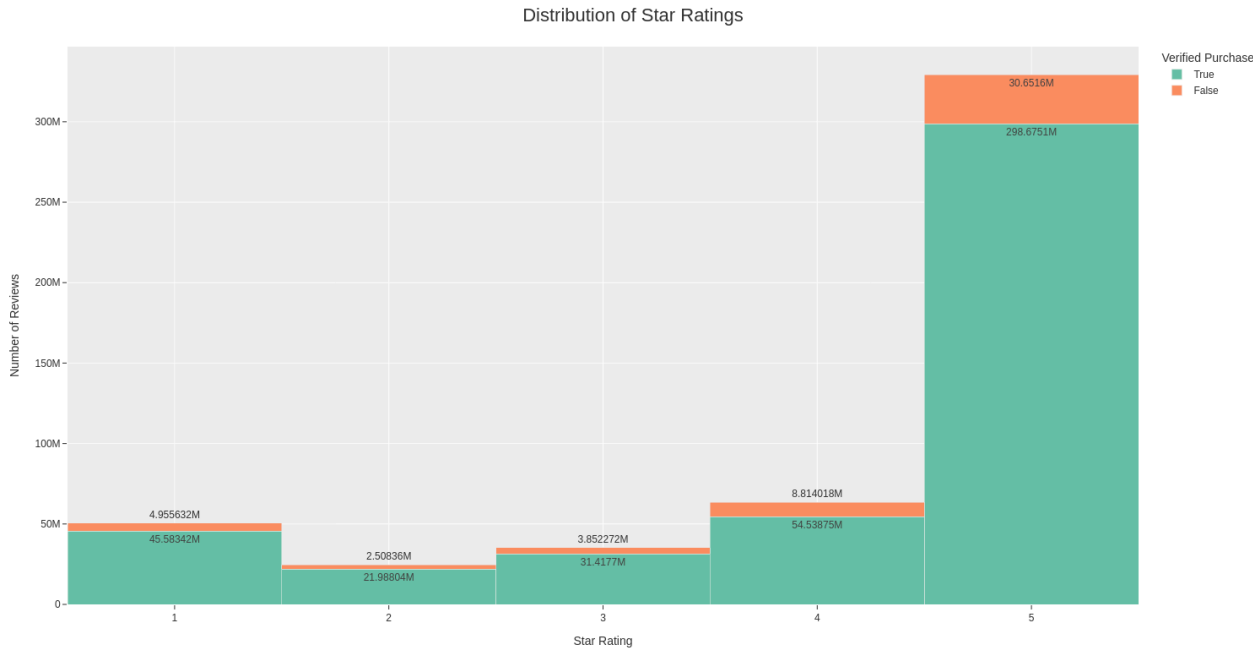
To conduct our exploratory data analysis, we leveraged **Polars'** lazy frame evaluation to select only the columns we needed, minimizing RAM usage. **Plotly** was chosen over **Matplotlib** for its interactive visuals, robust accessibility support, superior data interpretability and explainability, and as well as better scaling under high data loads as we leveraged all 502M rows in the dataset for completeness.

All charts were saved as PNG and PDF for persistence and review, then published online to enable permanent access and interactive **Plotly** features, see interactive plots (here). Furthermore, core figures were supplemented with additional visuals to enhance explainability.

To communicate data scale more effectively, we used abbreviations like **K** (thousand), **M** (million), and **B** (billion). This helps reduce numerical blindness from large numbers and improves explainability for non-statistical readers.

More detailed statistical insights of the plots are available in the attached Jupyter notebook and all plots can also be viewed in your web browser.

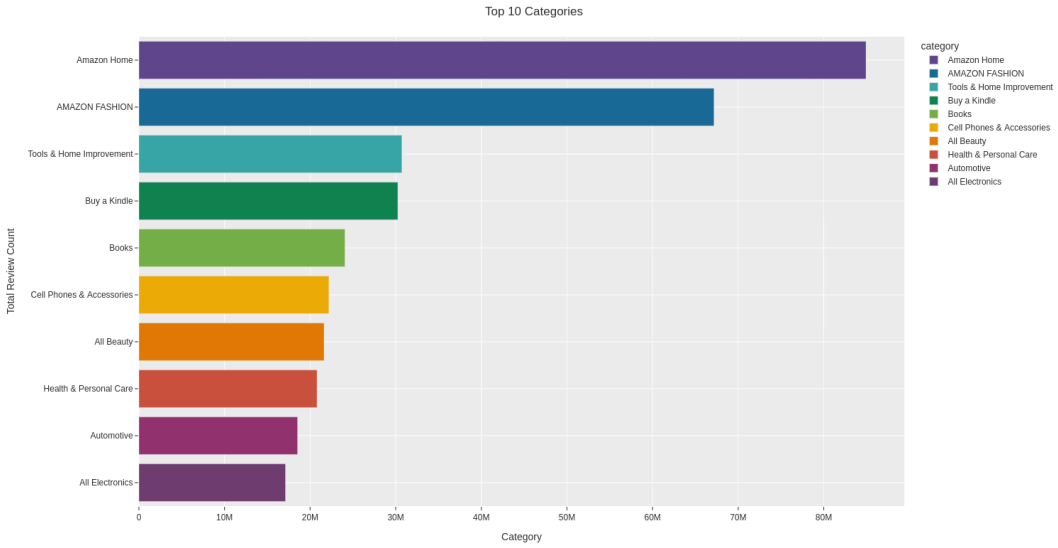
Star Rating Histogram



Histogram

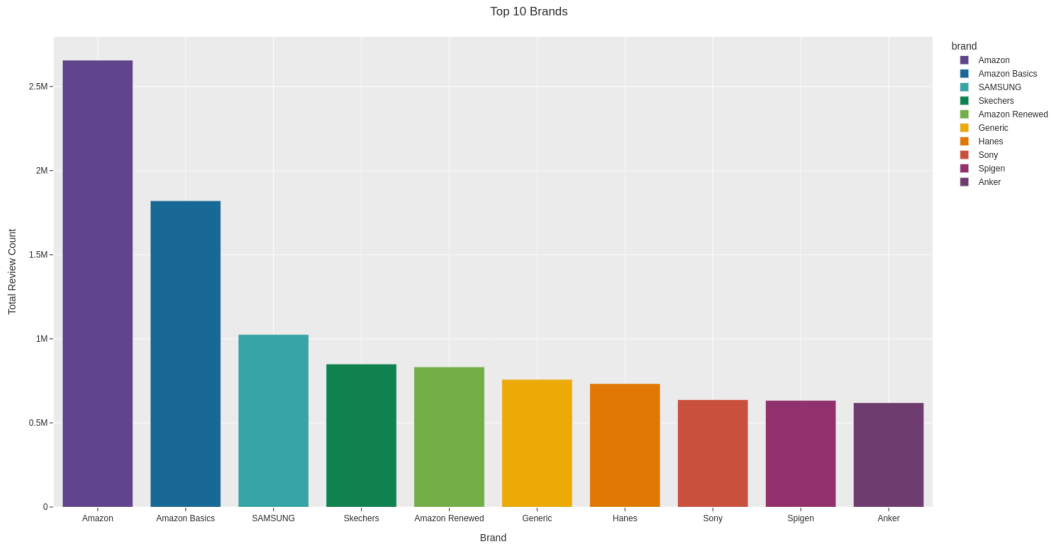
The data shows a strong positive sentiment in star ratings, with 75% being 4-5 stars and 5-star reviews alone making up 62.4%. Most 5-star ratings (90.7%) come from verified purchases, highlighting high satisfaction among confirmed buyers. Negative reviews (1-2 stars) are much less common at 15%.

Bar Charts



Categories

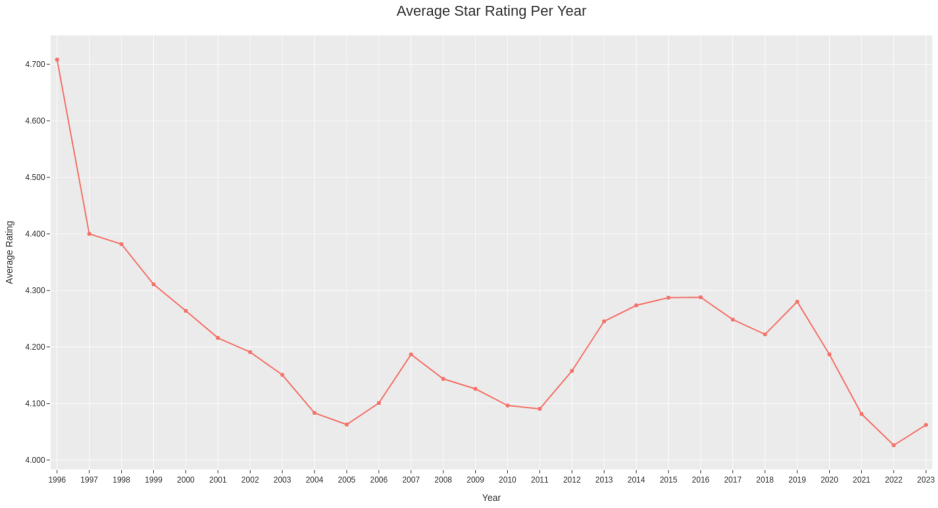
Review volume is heavily concentrated in **Amazon Home** and **Amazon Fashion**, which together account for over 45% of the top 10 total and are more than one standard deviation above the mean. The remaining categories reflected a positive skew and Pareto-like distribution where the categories cluster into mid (around 30M) and lower tiers (17-24M). This suggests that broad-appeal categories drive most of the reviews, while the categories beyond the top two offer diminishing returns.



Brands

The chart shows **Amazon** with 2.65M reviews (25.5%) and **AmazonBasics** with 1.82M (17.5%), together making up over 43% of the total. **SAMSUNG** follows with 1M (10%), while the next five brands range from 600k to 880k reviews (6–9%), underscoring **Amazon's** lead and a sharp drop-off after its brands.

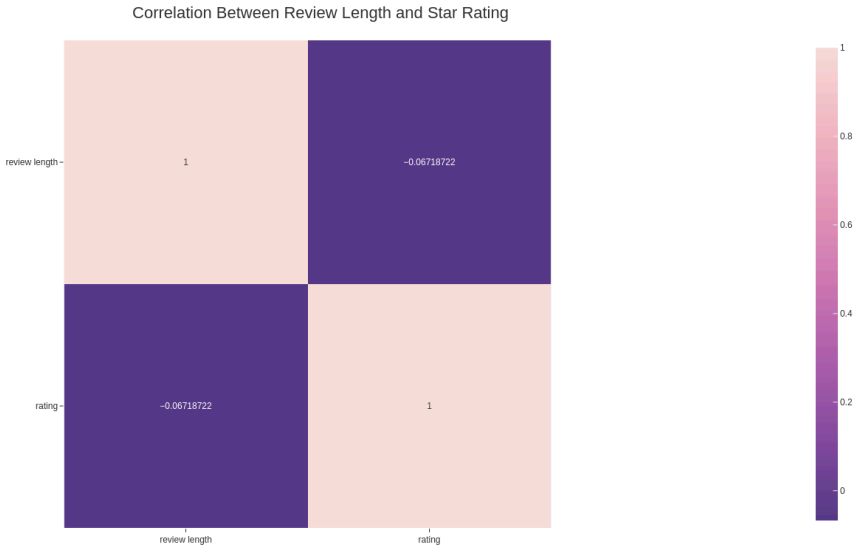
Line Charts



Star Ratings

From 1996 to 2023, average ratings fell **13.7%** from 4.71 to 4.06 by 2005, rebounded **5.5%** to **~4.29** in 2015–16, dipped to 4.03 in 2022, then edged up to 4.06 in 2023. Over the 28-year span, the mean rating was 4.21 (**SD = 0.14, CV ≈ 4%**) within a narrow **4.028–4.708** band; each phase change exceeded **2 × SD**, signaling genuine shifts in reviewer sentiment, although early data sparsity may inflate the 1996 high.

Correlation Matrix



Corr Matrix

The matrix shows a weak negative relationship between **length** and **star rating**. Meaning that **longer reviews** tend slightly towards **lower stars**, however, the correlation is so small, review verbosity offers virtually no predictive power of rating.

Binary Sentiment (Logistic Regression) (Task 4)

The ratings were binarized, with scores above 3 labeled as positive (1) and scores of 3 or below labeled as negative (0), converting the problem into a binary classification task. An 80/20 train-test split was then applied to the full five-hundred-million row dataset.

To accelerate processing, the `sklearnex` library was used. Sklearnex is an Intel extension that optimizes scikit-learn's performance. Stop words were also removed to enhance model performance and reduce memory usage.

For text preprocessing, the review text was lowercased and tokenized based on whitespace and punctuation. The TF-IDF vectorizer was used, with tokens that appeared in fewer than five reviews or in more than 80% of the reviews discarded, in order to eliminate rare and overly common terms.

A logistic regression classifier was used with default hyperparameters, except for `max_iter`, which was explicitly set to 100 to avoid runtime convergence warnings. To optimize memory usage, **Polars'** Lazy API was employed, selecting only the "Rating" and "Review Text" columns during data loading.

The model performed well, achieving an accuracy of 0.87685, precision of 0.897167, recall of 0.949722, and an F1 score of 0.922697. The classification results included 13,415,976 true positives, 7,965,617 false positives, 3,679,098 false negatives, and 69,496,368 true negatives.

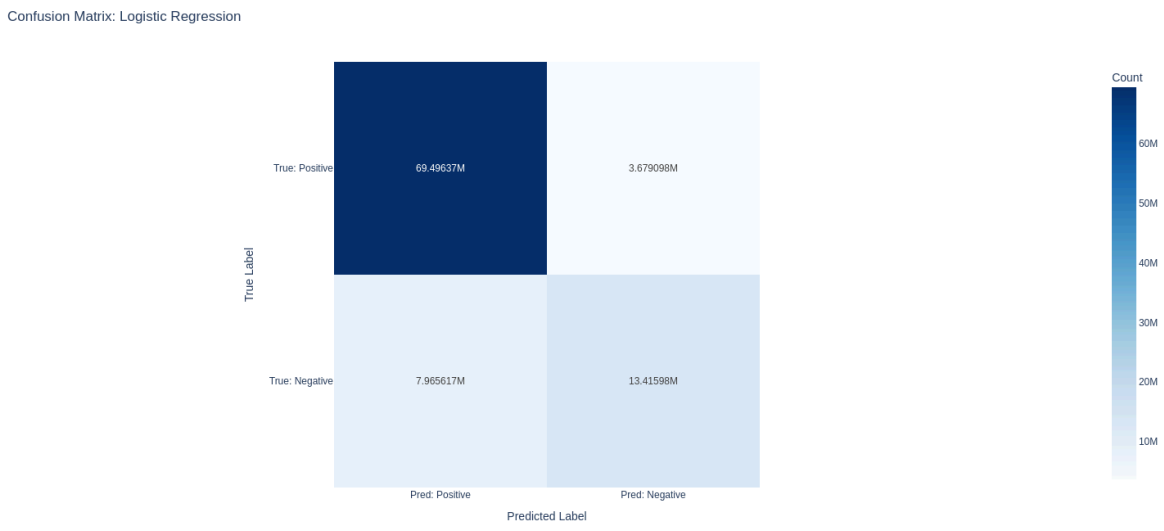


Figure 3: Logistic Regression Confusion Matrix

Confusion Matrix

Recommender (ALS) (Task 5)

A collaborative-filtering model was built by first loading only the `(user_id, asin, rating)` columns from the full dataset via **Polars'** Lazy API, filtering out users with fewer than five reviews, and encoding both IDs as contiguous integer indices. The remaining interactions were split 80/20 into train and test sets, from which a CSR sparse matrix was constructed.

`Implicit` was chosen over PySpark MLlib for its in-process Cython implementation, which runs efficiently on low-compute environments, integrates seamlessly with Polars-derived sparse matrices (avoiding JVM serialization), and enables straightforward hyperparameter tuning with optional GPU acceleration, all with significantly less infrastructure overhead than managing a Spark cluster. Since training was conducted on a virtual machine, avoiding the setup and complexity of a JVM further reinforced this choice. To summarize, `implicit` was chosen as it better integrates with our workflow on low compute.

ALS was then applied using the `implicit` library (50 factors, λ=0.01, 15 iterations), and test-set predictions were obtained by dot-product of user and item factors; the resulting RMSE was calculated 4.4181.

The high RMSE resulted from the dataset's large size and low interaction variance, and assignment constraints on feature selection and minimal tuning prevented further optimization. Despite top-5 recommendations scoring over 4.0 for three users, individual estimates remain imprecise due to the high RMSE.

```
User AECVLWXQSOTOA4R7IB4KWTUB4H3Q (idx=3970955):
  ASIN B01MFGX5GI – pred. rating 5.00
  ASIN B07Q5TL9SQ – pred. rating 4.78
  ASIN B06Y1264PX – pred. rating 4.76
  ASIN B00K2EOONI – pred. rating 4.73
  ASIN B07MV8SWZF – pred. rating 4.68

User AFTKJPWKJIC7R3J23RVC5JYPUJGA (idx=876054):
  ASIN B00DS842HS – pred. rating 5.00
  ASIN B00FLYWNYQ – pred. rating 4.44
  ASIN B0009X29WK – pred. rating 4.36
  ASIN B00016XJ4M – pred. rating 4.16
  ASIN B0026HDURA – pred. rating 4.14

User AEJPJARKMDB6YBVFJDWYGGONXV4Q (idx=10046557):
  ASIN B079QHML21 – pred. rating 5.00
  ASIN B001T7QJ9O – pred. rating 4.50
  ASIN B0043T7FXE – pred. rating 4.49
  ASIN B004S8F7QM – pred. rating 4.37
  ASIN B003NR57BY – pred. rating 4.12
```

Figure 4: Top 5 Recommendations & Predicted Rating for Random Users

Clustering (k-means) (Task 6)

K-means clustering was used to segment products into five groups based on four features: mean user rating, total number of reviews, brand ID, and category ID. These features were derived by aggregating the entire review dataset, ensuring that only users with at least five reviews were included to reduce noise. Brand and category names were encoded as unique integer IDs to make them compatible with the algorithm.

Clustering was run using default settings until convergence. After assigning products to clusters, each group was analyzed by calculating its size and the average of each feature. This helped identify distinct product segments - for example, clusters of frequently reviewed, high-rated products likely from major brands, or clusters consisting of low-rated items with minimal user engagement and unknown categories.

Analysis offered insight into the structure of the product landscape and potential patterns in user interactions. **Cluster 0**, the largest (4.9M items), sits at a middling 4.099 rating with 13.11 reviews, suggesting broad but tepid enthusiasm.

Cluster 1 (1.5M items) grouped high-rated books (avg. 4.32) with the lowest review count (avg. 6.13), indicating a small but loyal audience. **Cluster 2** (1.8M items) specifically contained Kindle books with average ratings (4.28) and reviews (7.7) characteristic of mid-tier bestsellers.

Cluster 3 (2.5M items) strikes a balance of reviews and ratings (avg. 4.20 rating, 10.5 reviews), reflecting steady engagement from established brands. **Cluster 4** (2.4M items) captured fashion items notable for the highest average reviews (15.8) but lower ratings (avg. 4.08), underscoring that mass exposure often dilutes quality perception.

From a statistical perspective, an inverse relationship between brand prominence and review volume appears to emerge. The highest-rated clusters are the smallest and most niche, while the largest, more generic segments trade premium appeal for sheer scale, suggesting that quality does not scale linearly with popularity.

shape: (5, 6)

cluster	cluster_size	avg_mean_rating	avg_total_reviews	avg_brand_id	avg_category_id
i32	u32	f64	f64	f64	f64
0	4900703	4.098899	13.111658	656981.198234	185631.833606
1	1520795	4.324956	6.132356	4.0627e6	195285.389854
2	1820326	4.279179	7.70994	2.7330e6	193409.32291
3	2569403	4.201071	10.548661	1.5657e6	190014.075359
4	24554101	4.077346	15.8126	81107.899496	182308.524842

Figure 5: K-means Clustering

Conclusion

Resources

Plotly was selected over **Matplotlib** due to its superior scalability and built-in accessibility features, which enhance the clarity and interactivity of visualizations. For data processing, we opted for **Polars**, as previously noted, because it consistently outperforms both **PySpark** and **Pandas** right out of the box and especially at scale.

Sklearnex was chosen to accelerate scikit-learn-based algorithms, offering significant speed improvements with Intel CPUs with no code changes. For Alternating Least Squares (ALS), we opted for the **Implicit** library over **Spark's** implementation primarily to avoid the need for **JVM** installation in virtualized environments. In addition to this streamlined setup, Implicit also offers better runtime performance and reduced memory overhead compared to the more resource-intensive, **JVM-based Spark**.

Hardware Specification

Ubuntu Virtual Machine with: Intel Xeon E5-2690 CPU, 64GB of 2400MHz ECC RAM with an additional 100GB of dedicated swap space and 300GB 3D NAND SATA storage.

Challenges

Some **DataFrame** manipulations for plotting were compute-intensive, consuming significant memory and RAM and taking a long time to process. To reduce wait times during subsequent runs, the manipulated **DataFrames** were saved as Parquet files for reuse.

Even though the caching directory was explicitly set to another drive, the system continued to cache on the home drive as well, resulting in *two* separate caches and doubling the space usage. A delete function was made to clear the cache after every subsequent download to free up space.

Initially, **Azure** was the chosen compute option, but the best configuration offered under the education edition only included 64GB of RAM, which was the same as what was available locally. **DigitalOcean** was also considered, but it faced similar limitations as **Azure**.

Although higher-compute **Azure** machines were eventually identified, switching platforms at a late stage proved impractical due to the time required to reacquire and process the data (more details). Limited SSD storage also posed potential challenges. Additionally, the availability of virtual machines fluctuated frequently—VMs that were once accessible could become unavailable without notice (more details).

Code Repository

GitHub Repository

carrot2803/comp3610A3

github.com/carrot2803/comp3610A3

Visualizations

github pages

carrot2803.github.io/comp3610A3

Imgur Album

imgur link

imgur.com/a/comp-3610-assignment-3-2iUDUBa

PDF/PNG

carrot2803/comp3610/data/processed

github.com/carrot2803/comp3610A3/data/processed