Joel Kennerley 46097032

Data 301 Project Final Report

Which top 5 pairs of highly-rated books most strongly predict the presence of another highly-rated book?

Abstract / Summary

For this project, I will be using the Goodreads interactions comics graphic dataset, and the good reads comic graphic metadata data set. The research question I will explore is: Which top 5 pairs of highly-rated books most strongly predict the presence of another highly-rated book? The algorithms I will use are groupby, and the A-Priori algorithm, I will also use association analysis to determine the confidence of book pairs. The intended result is a ranked list of the highest confidence scores for pair rules. The significance of this question lies in identifying common book preferences among users, which can enhance recommendation systems and help target marketing strategies.

Introduction

Background:

The interaction dataset contains user reviews from Goodreads, where each record represents a user's review of a book. The features I will be using from this dataset are ‘user ID’, ‘Book ID’, ‘Rating’, and ‘is read’. The user id is an id unique to every user. The book id is an id that is unique to every book. Rating is the rating out of 5 that the user rated the book. And ‘is read’ is a Boolean that shows whether a user has read the book. This is important as users are able to leave reviews even if they have not read a book. The comic graphic metadata dataset contains all metadata related to each book. I am using this as it contains ‘book id’ and ‘title’, which I need to retrieve titles of books after finding the strongly predicted books.

To identify patterns in user behavior, I will use association rule mining, specifically the Apriori algorithm. In this approach, each user is treated as a "basket," and the books they have reviewed are the items in that basket. The Apriori algorithm is designed to find frequent itemsets, combinations of books that appear together across many users' baskets. It does this by first identifying individual books that meet a minimum support threshold, then combining these into larger itemsets (such as pairs or triplets), and pruning those that do not meet the threshold. The resulting bag of itemset counts then helps me calculate the confidence scores for each pair of books. Confidence measures how likely it is that a user who reviewed one book also reviewed another, providing insight into the strength of the association between items.

Motivation:

As an avid reader, I’ve often found myself exploring Goodreads to discover books based on user reviews and ratings. This project brings together my interest in literature and the data mining techniques I learned in DATA301, allowing me to explore patterns in readers' preferences and share insights that could help others discover new books based on shared tastes.

Research Question:
This research question is relevant because the dataset includes user-generated reviews and ratings for a wide range of books, allowing us to analyze patterns in positive user preferences. By filtering for reviews with ratings above 4, we focus specifically on books that users enjoyed. Applying the Apriori algorithm enables the discovery of frequent itemsets, pairs of books that commonly appear together in users' highly-rated reviews. From these itemsets, we can calculate confidence scores to determine how strongly the presence of two books predicts a third. This helps reveal patterns in user behavior and can provide valuable insights for personalized book recommendations or understanding reader preferences.

Design and Methods:

To accomplish the objective of identifying which top 5 pairs of highly-rated books most strongly predict the presence of another highly-rated book, I applied the Apriori algorithm. This algorithm is widely used in association rule mining to find frequent itemsets in transactional datasets. In this project, each user's list of highly-rated books (ratings above 4) is treated as a transaction, where each book represents an item in their basket. The Apriori algorithm first identifies frequent itemsets, combinations of books that often appear together across different users' baskets. From these itemsets, association rules of the form {Book A} -> Book B are generated, and confidence scores are calculated. These scores reflect how likely it is that users who highly rate Book A also highly rate Book B. The top 5 rules with the highest confidence scores are selected to answer the research question.

The data flow begins with importing and preprocessing the Goodreads dataset, including filtering out reviews with ratings of 4 or below and transforming the data into a transactional format where each transaction contains the set of books highly rated by a single user. After cleaning and structuring the data, the Apriori algorithm is applied to extract frequent itemsets of size two. Confidence scores are calculated for these rules, and the results are sorted to identify the top 5 with the highest predictive power. The implementation is done in Python, using dask for efficient parallel processing of the dataset. The project was executed in three stages: data preprocessing in week one, algorithm implementation and rule evaluation in week two, and result analysis and reporting in week three.

Libraries I used in this project

- Dask - Used dask bags to parallelize data processing and handle large datasets efficiently. I used methods map, foldby, filter, repartition, pluck, flatten, frequencies, and join.
- Json - Used to load the json data into a dask bag.
- Time - Used to time my program with different sizes and processors.
- Sys - To access command-line arguments to specify number of processors and file size when testing on google cloud.
- Multiprocessing - Used to access the number of cpus being used.
- os - Used to retrieve stats on a file for good cloud use.
- urllib.request - Used to download my datasets off of google storage buckets.
- Gzip - Used to unzip the json.gz files.

- Itertools - Used islice to only read in subsets of my data for testing on the google cloud.

```python
import dask.bag as db
import json
import gzip
import shutil
import multiprocessing
import os
import urllib.request

# downloading json.gz files from google cloud buckets
url = 'https://storage.googleapis.com/jsonstuff/goodreads_interactions_comics_graphic.json.gz'
filename = 'goodreads_interactions_comics_graphic.json.gz'
urllib.request.urlretrieve(url, filename)

# unzipping json.gz file
with gzip.open(filename, 'rb') as f_in:
  with open('goodreads_interactions_comics_graphic.json', 'wb') as f_out:
    shutil.copyfileobj(f_in, f_out)

# reading file into dask bag
filename = 'goodreads_interactions_comics_graphic.json'
n_cpus = multiprocessing.cpu_count()
file_stats = os.stat(filename)
file_mb = file_stats.st_size / (1024 * 1024)
b = db.read_text(filename, blocksize=str((file_mb / n_cpus))+"MB")
parsed_bag = b.map(json.loads)
```

I begin by importing all the necessary libraries. Next, I download the interactions dataset from a Google Cloud Storage bucket, unzip the file, and load it into a Dask bag for further processing.

```python
def clean_data(bag):
  bag = bag.map(lambda x: (x['user_id'], x['book_id'], x['is_read'], x['rating']))
  bag = bag.filter(lambda x: x[3] >= 4 and x[2] is True)
  return bag
```

Here, I use the Dask bag map function to filter the dataset and retain only the necessary features. I then apply additional filters to keep only books with high ratings (4 or 5) and those that have been read by users.

```python
def transform_records(bag):
  n_cpus = multiprocessing.cpu_count()
  user_books = bag.foldby(key=lambda x: x[0], binop=lambda acc, x: acc + [x[1]], initial=[], combine=lambda acc1, acc2: acc1 + acc2 )
  user_books = user_books.repartition(n_cpus)
  user_books = user_books.pluck(1)
  return user_books
```

The purpose of this function is to transform the data so that each entry in the bag represents a user along with all the books they have reviewed, e.g., [[‘32’,’4’],[‘1’,’2’,’54’,]...]. This is achieved using the foldby operation, which groups entries by user and aggregates their reviewed books into a single list. Since foldby results in a bag with only one partition, I then repartition the bag to improve parallelism and performance

```
def a_priori_step1(user_books_bag):
  item_counts = user_books_bag.flatten().frequencies()
  return item_counts
```

This function takes the bag, flattens it into individual book entries, and calculates the frequency of each reviewed book.

```
def a_priori(text_file_bag, support=2000):
  item_counts = text_file_bag.flatten().frequencies()
  significant_item_counts = item_counts.filter(lambda x: x[1]>=support)
  significant_set = set(significant_item_counts.map(lambda x: x[0]).compute())

  def generate_frequent_pairs(books):
    pair_list = []
    for i in range(len(books)):
      if books[i] in significant_set:
        for j in range(i+1, len(books)):
          if books[j] in significant_set:
            pair_list.append(tuple(sorted((books[i], books[j]))))
    return pair_list
  pair_counts = text_file_bag.map(generate_frequent_pairs).flatten().frequencies()
  #pair_counts = pair_counts.repartition(n_cpus)
  return pair_counts
```

This is an implementation of the Apriori algorithm designed to find frequently co-reviewed book pairs. It starts by computing the frequency of individual books across all users and filters out those with a frequency below a given support threshold (set to 2000 for the Comics and Graphic dataset). These frequent books are stored in a set to allow for fast membership checks.

Next, a nested function is defined to generate candidate book pairs from each user's list of reviewed books. For each book in the list, the function checks whether it meets the support threshold. If it does, it then checks all books that appear after it in the list, again filtering out those below the threshold. This approach avoids redundant checks and ensures that each pair is considered only once. The function returns a list of qualifying book pairs for that user.

The main algorithm then applies this function to every user's book list using a map operation over the Dask bag. The resulting lists of pairs are flattened into a single bag of individual (book1, book2) tuples. Finally, the algorithm calculates the frequency of each pair using frequencies(), identifying the most commonly co-reviewed pairs.

```
def confidence(item_counts, pair_counts):
  split_pairs = pair_counts.map(lambda x: [(x[0][0], (x[0][1], x[1])), (x[0][1], (x[0][0], x[1]))]).flatten()
  joint = split_pairs.join(item_counts, lambda x: x[0])
  output = joint.map(lambda x: ((x[0][0],x[1][1][0]), x[1][1][1]/x[0][1]))

  return output
```

This function calculates the confidence of pairwise association rules. It begins by mapping each frequent pair into two directional tuples, effectively separating each book in the pair. For example: ((book1, book2), count) -> [(book1, (book2, count)), (book2, (book1, count))].

These are then flattened into a single bag. This step is necessary because confidence is not symmetric, that is, the rule book1 -> book2 is not the same as book2 -> book1. Confidence is defined as: sup(AnB)/sup(A). To calculate this, we join the pair counts with the individual item counts (sup(book)). Finally, we apply a map function over all pair rules to calculate the confidence of each directional rule.

```
def load_book_metadata(jsongz):
  book_bag = db.read_text(jsongz).map(json.loads)
  titles = book_bag.map(lambda x: (x['book_id'], x['title']))
  return titles
```

This function loads the metadata dataset and filters it to retain only the information needed to extract book titles. The ‘jsongz’ parameter refers to the URL of the dataset hosted in a Google Cloud Storage bucket. From this data, a Dask bag is created and computed to generate a dictionary of titles: title_dict = dict(titles.compute()). While this step is computationally expensive, since it involves materializing the entire metadata dataset, I was unable to find a more efficient approach within the available time frame.

```
def get_titles_for_pair_fast(item):
    (book_id_1, book_id_2), count = item
    title1 = title_dict.get(book_id_1, "Unknown")
    title2 = title_dict.get(book_id_2, "Unknown")
    return ((title1, title2), count)
```

```
top_titles = results.map(get_titles_for_pair_fast)
```

This function is then applied to the top 5 pairs with the highest confidence scores, retrieving the corresponding book titles.

Results:

Supporting graphs and figures:

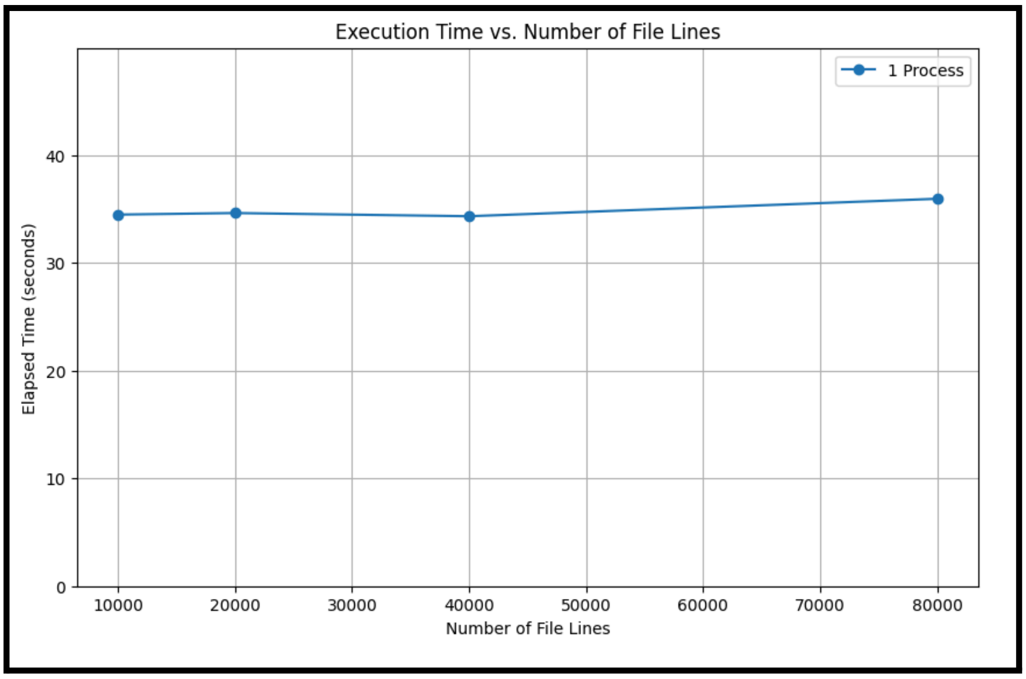


Figure 1: program time vs file size

```
elapsed_times_1_process = [34.48950982093811, 34.634485721588135,
34.33740568161011, 35.958076715469361]
```

Figure 1 above represents the scalability of my program on a single processor. I tested the program with the file having 10,000, 20,000, 40,000, and 80,000 lines in the good reads interaction comics graphic dataset. In the graph we can see that the time to run the algorithm stays reasonably constant with a very minor increase in time from 40,000 to 80,000 lines. This shows that the quantity of data has little effect on the efficiency of the program. After further analysis my program has a bottleneck when creating a dictionary that matches book ids with their titles, this is because I am calling:

```
title_dict = dict(titles.compute())
```

This causes the comics_books_graphic_comic.json file to compute the whole dataset. The size of this file never changes because when taking subsets of the interactions data set it does not specify which books will be included in the interactions data. This costs an average ~ 30 seconds each time the program is run regardless of the number of processors and the size of the file.

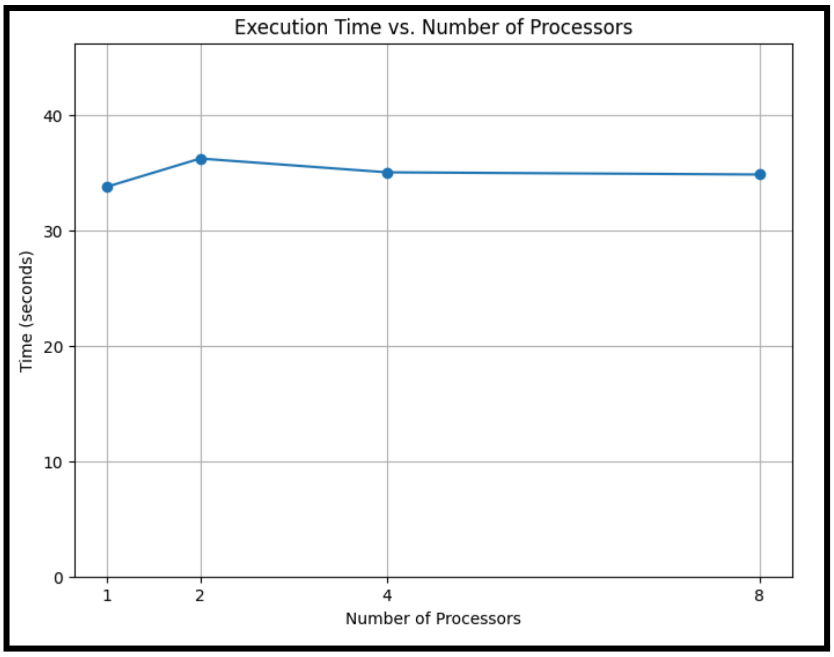


Figure 2: Program time vs number of processors

```
times = [33.794820070266724, 36.23728942871094, 35.043009519577026,
34.85246181488037]
```

Similar to the single processor scalability we see the same bottleneck impacting the program runtime testing the weak scalability as seen in Figure 2.

Because of this I have decided to treat it as a constant and measure the runtime excluding the bottleneck. In the new graphs I also use subsets with 200,000, 400,000, 800,000, and 1,600,000 lines. I have increased the file size as the data that is being passed through to my main algorithm will be much smaller due to filtering and transformation.

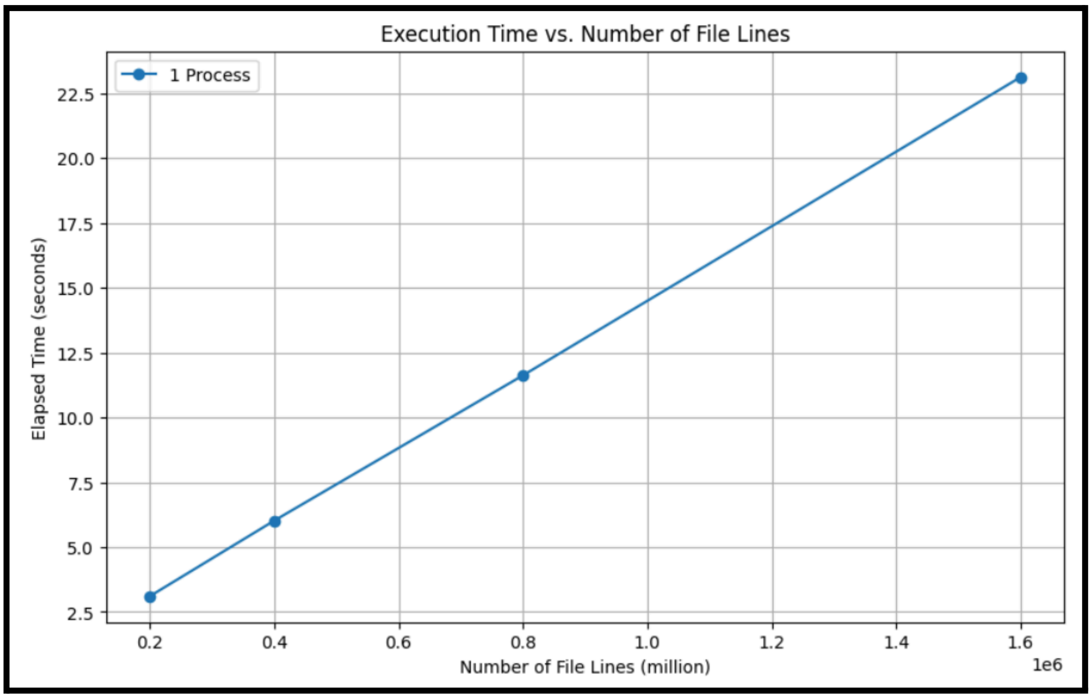


Figure 3: Program runtime vs file size (excluding expensive bottleneck)

```
time_1_process = [3.1081, 6.0202, 11.6211, 23.1151]
```

In Figure 3 ,when using a single processor on different file sizes we can see that the increase in execution time with respect to increase in file size is linear which is what we would expect to see.

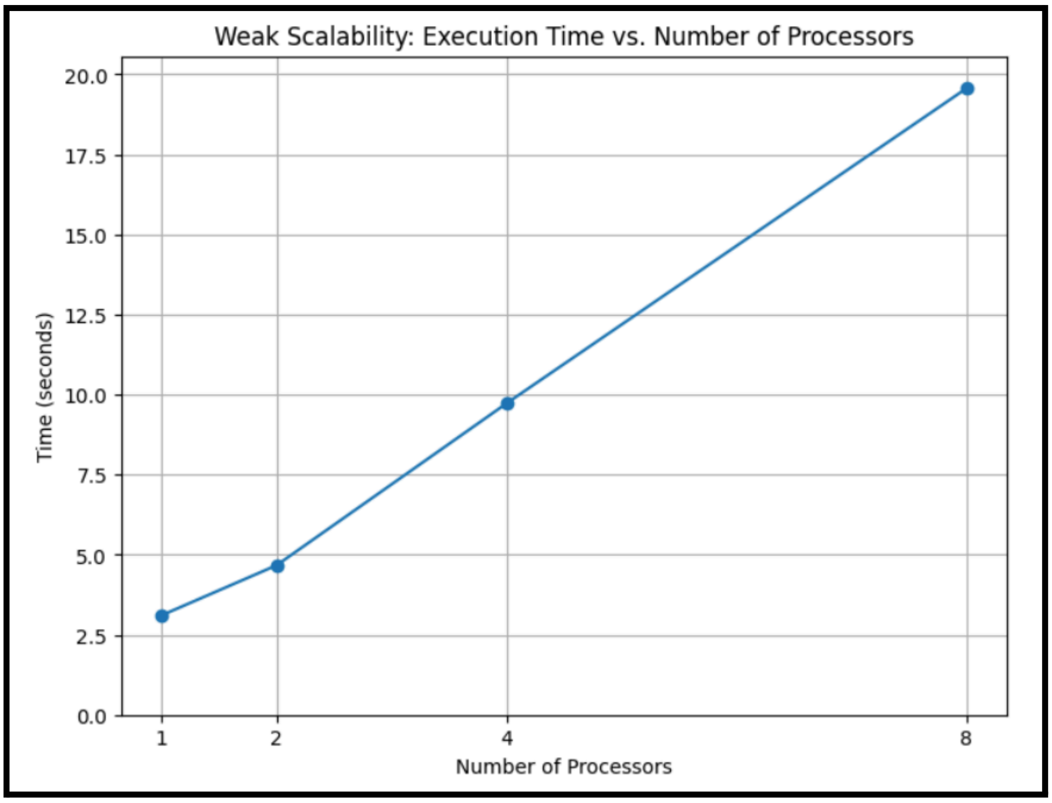


Figure 4: Program time vs number of processors (excluding expensive bottleneck)

```
times = [3.1081, 4.6712, 9.7280, 19.5664]
```

Figure 4 shows the weak scalability of the program. Similar to the single processor, the relationship still seems to be linearly increasing, suggesting that this program has weak scalability. This could be due to large overhead and communication costs between workers, or poor utilisation of parallelism in the program.

Which top 5 pairs of highly-rated books most strongly predict the presence of another highly-rated book?

```
[(('Death Note, Vol. 2: Confluence (Death Note, #2)', 'Death Note, Vol.
1: Boredom (Death Note, #1)'), 0.934113712374582), (('Death Note, Vol.
4: Love (Death Note, #4)', 'Death Note, Vol. 3: Hard Run (Death Note,
#3)'), 0.9303043670048522), (('Death Note, Vol. 4: Love (Death Note,
#4)', 'Death Note, Vol. 1: Boredom (Death Note, #1)'),
0.9303043670048522), (('Death Note, Vol. 5: Whiteout (Death Note, #5)',
'Death Note, Vol. 1: Boredom (Death Note, #1)'), 0.9297029702970298),
(('Death Note, Vol. 3: Hard Run (Death Note, #3)', 'Death Note, Vol. 2:
Confluence (Death Note, #2)'), 0.928766056831452)]
```

The top five pairs of highly-rated books that best predict another highly-rated book all come from the *Death Note* series. The strongest pair is *Death Note, Vol. 2: Confluence* and *Vol. 1: Boredom*, with a confidence score of 0.934, meaning that 93.4% of users who rated Volume 2 highly also rated Volume 1 highly. This trend continues across the other top pairs, showing strong overlap in who reads and enjoys these books.

This suggests that *Death Note* fans are dedicated to the series and tend to read multiple volumes. This is useful for recommendation systems because if someone rates one volume highly, there's a good chance they'll enjoy the others too. It also gives a strong case for promoting the books together or offering them as a bundle.

Conclusion:

I was able to answer the research question by mining frequent itemsets and generating association rules based on high user ratings. By calculating the confidence of rules in both directions (A -> B and B -> A), I identified the top 5 pairs of highly-rated books that strongly predict the presence of another. However, all the top pairs were from the same series *Death Note* which suggests the rules reflect user loyalty within a single series rather than broader reading patterns. So while the results are accurate, they may be limited in scope for recommending books across different series.

One implication of my results is that they may not be as useful for recommending new books to users. This is because if we were to recommend a user *Death Note, Vol. 1: Boredom* based on the fact they reviewed *Death Note, Vol. 2: Confluence*, it would not provide much value as users would typically read *Vol. 1* before reading *Vol. 2*. This is a common theme as the other highest confidence rules all also are formatted in a way that it is very likely for them to have read the older book first.

Future directions I would take with this project include exploring itemsets with more than two books. This would be useful if I wanted to investigate whether selling specific book bundles could be a viable strategy. It could reveal patterns like sets of three or more books that users often rate highly together. Another useful extension would be to calculate lift scores alongside confidence. Lift would help assess how much stronger the association is compared to what we'd expect if the books were rated independently, giving a clearer picture of which associations are genuinely interesting.

Critique of Design and Project:

One part of my design that could have worked better with a different approach is the use of the A-Priori algorithm. While it successfully identified frequent pairs of books reviewed by users, it did not provide particularly useful insights for recommending new books. Many of the top results were books from the same series, such as *Death Note, Vol 1* and Death Note, *Vol 2*. These results aren't very meaningful because users are likely to read earlier volumes before later ones. Recommending Vol 1 based on someone reviewing Vol 2 doesn't add much value, since the user probably already read it. This pattern was common across the top pairs, which made the results predictable and not especially helpful for real recommendation purposes.

A better approach could have been to use cosine similarity on a user-book matrix. Instead of focusing on which books commonly appear together, this method compares how similar books are based on user review patterns. It would have allowed for recommendations based on shared user preferences, rather than just co-occurrence. This way, the model could suggest books outside a direct series, offering users new and relevant reading options. Cosine similarity would likely have led to more varied and meaningful recommendations.

Reflection:

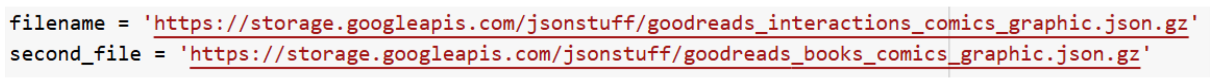
I found course concepts like parallelism, map reduce, and association analysis really useful for completing this project. Tools such as the Dask API were also helpful when writing and running the code. This project taught me how important parallelism is when working with large datasets and helped me become more confident using Dask in practice. It also gave me a better understanding of how association analysis works and how it can be applied to real-world problems. On top of that, managing the different stages of the project improved my time management and planning skills.

References:

Item recommendation on monotonic behavior chains
Mengting Wan, Julian McAuley
RecSys, 2018

```
book_bag = db.read_text(jsongz).map(json.loads)
```

I got this code from ChatGPT to load in the data.



```
filename = 'https://storage.googleapis.com/jsonstuff/goodreads_interactions_comics_graphic.json.gz'
second_file = 'https://storage.googleapis.com/jsonstuff/goodreads_books_comics_graphic.json.gz'
```

I also used ChatGPT to help me put my data files into buckets, so that I can access the data using google cloud for testing the parallelism of my project.

Also used labs 3, and 5 as a base for using google cloud.