Market-Basket Analysis

Data Science for Economics - Algorithms for Massive Data

Melissa Rizzi - Angelica Longo
melissa.rizzi@studenti.unimi.it
angelica.longo@studenti.unimi.it

April 2025

1 Introduction

Market-basket analysis is a widely used technique in data mining to identify frequent itemsets—groups of items that are often purchased together. The goal of this project is to implement a system capable of detecting frequent itemsets using the Amazon Books Reviews dataset, publicly available on Kaggle under the CC0 license. It is retrieved dynamically through an API during the script execution and contains user-generated book reviews, including ratings and user identifiers.

The raw dataset contains 3 million rows and 10 columns. To process it efficiently, PySpark is used, a library that leverages the MapReduce paradigm to handle large-scale computations while ensuring fault tolerance, significantly speeding up data processing compared to traditional approaches.

2 Data Cleaning

To ensure optimal model performance, a thorough data-cleaning process is carried out as an initial step, minimizing errors and enhancing the accuracy and reliability of the experiment. For the purpose of this analysis, only relevant variables are selected:

1. **User_id**: Unique identifier for each user.
2. **Id**: Unique identifier for each book.
3. **Review/Score**: Numerical rating given by users to books.

Furthermore, data integrity checks are performed by ensuring that all score values fall within the range of 1 to 5.

2.1 Handling Missing Values

A significant number of missing values are observed in the `User_id` field. One possible explanation is that users who leave reviews without registering on the platform do not have an associated user ID. Since the analysis requires tracking user purchases to identify baskets, the absence of this key variable prevents the analysis from proceeding.

An alternative approach is considered: using profile names instead of user IDs by assigning an identifier to users with the same name. However, this method is unreliable due to name duplication, and it was further invalidated by the fact that profile names had even more missing values than user IDs. Given these constraints, all rows with missing User_id values are dropped.

2.2 Removing Duplicates

The next step is to handle duplicate entries. First, exact duplicates across all columns are removed. The next step is to handle duplicate entries. First, exact duplicates across all columns are removed. Then, duplicate entries based only on the `Id` and `User_id` fields are identified. If a user has rated the same book twice, the average of the two scores is calculated, and a new column is created to store this average score for each user-book pair.

2.3 Filtering Useful Rows

Before creating the used subsample, additional filters are applied:

- **Filter by score:** only ratings greater than 2 are retained. The rationale behind this choice is that a low rating typically indicates a book the user has read but does not consider part of their ideal collection. This helps ensure that collaborative filtering recommends items that are liked by other users with similar tastes.
- **Filter by user activity:** users who have rated only one book are excluded, as they do not provide meaningful baskets for analysis.

A brief summary of all the performed steps can be found in the following Table 1.

Processing Step	**Rows**
Initial Dataset	3,000,000
After Removing Missing Values	2,420,237
After Removing Exact Duplicates	2,383,199
After Removing User-Book Duplicates	2,380,155
Filtering for Ratings >= 3 and at least 2 Items Reviewed	1,488,780

Table 1: Dataset size after each processing step

3 Subsampling for Efficient Processing

To optimize computation and ensure efficient processing, a representative subsample is created using PySpark, which enables the handling of large datasets in a scalable and efficient manner. Built on top of Apache Spark, PySpark is designed to leverage distributed computing, allowing data to be processed in parallel across multiple machines in a cluster. This makes PySpark ideal for processing massive datasets that would be difficult or impossible to handle on a single machine.

PySpark's strength lies in its ability to perform large-scale data transformations and actions through Resilient Distributed Datasets (RDDs). These RDDs allow data to be distributed, processed in parallel, and kept fault-tolerant, allowing data recovery in case of failures, making it an ideal choice for large-scale analytics, machine learning, and big data processing. With PySpark, operations such as subsampling can be done efficiently even with large datasets, as computations are spread across multiple nodes in the cluster.

Rather than taking a completely random sample, a more structured approach is adopted:

- The dataset contains 268,637 unique users.
- Instead of randomly selecting rows, 20% of the users along with all their ratings is retained, ensuring that baskets remain representative for those selected users.
- To ensure the replicability of the experiment, a random seed is set, in order to always obtain the same set of users.

The final subsample is then composed by 295,872 rows, instead of the 1,488,780 of the whole dataset.

3.1 Subsample Scalability

To validate the consistency of this sub-sample with the original cleaned dataset and ensure that the algorithm results can scale correctly to larger datasets, the computation and comparison of the mean rating, standard deviation and the average number of books rated per user is performed. The considered measures in Table 2 are quite similar, which allows us to conclude that the sample is representative of the original dataset.

Dataset	**Mean Rating**	**Standard Deviation**	**Avg. Books per User**
Full Dataset	4.54	0.68	5.54
Subsample	4.54	0.68	5.49

Table 2: Comparison between the original dataset and the subsample

The structure of the final dataset that will be used for market-basket analysis is shown in the following Table 3. By analyzing this dataset, patterns in user behavior can be uncovered, with the goal to improve recommendation system.

Id	**User_id**	**Score**
0001047604	A1ZQ1LUQ9R6JHZ	5.0
0001047655	A12N9YU5K516JF	4.0
0001047655	A1EB4FLIXNX0LK	4.0
0001047655	A1NS4974T51EU1	5.0
0001047655	A2C8IVS3AEH96R	3.0

Table 3: Final Data

4 Algorithms Implementation

The goal is to identify sets of books that have been rated by a significant number of users while excluding individual frequent items. For a pair of books to be considered frequent, each book must first appear frequently on its own. Books rated by only a few users are excluded, shifting the focus to meaningful book combinations. The algorithm then progresses from frequent pairs to larger itemsets (triples, quadruples), ensuring that each subset of a larger itemset is also frequent before considering it. This step-by-step approach guarantees that only truly frequent itemsets are retained.

The first implemented algorithm is the A-Priori algorithm, which works by eliminating large candidate sets early in the process, first analyzing smaller sets. It relies on the principle that a large set cannot be frequent unless all of its subsets are frequent.

The second approach includes the SON (Savasere, Omiecinski, and Navathe) algorithm, which leverages parallelism, such as MapReduce-based formulations, to improve processing efficiency.

Before starting the implementation, a function is created to allow the user to choose between using the original preprocessed dataset or the subsample. This function takes a boolean parameter that determines whether to work with a sample or the entire dataset. It returns the chosen dataset and prints its size, providing an indication of the amount of data that will be processed.

In this case, the subsample is used to speed up the process, while at the end of the experiment, the algorithms will be re-trained on the original dataset to assess scalability.

4.1 A-Priori Algorithm

The A-Priori algorithm is an association rule mining method used to discover frequent patterns within large datasets. It works by first identifying the most frequently occurring itemsets and then extracting association rules that express relationships between these itemsets. The algorithm follows an iterative approach, progressively eliminating less frequent itemsets, improving efficiency.

The A-Priori algorithm can be implemented directly through a MLxtend function, which is not compatible with PySpark. Thus, both methods will be implemented: the A-Priori function using a Pandas dataset, as well as a custom implementation of the algorithm built in PySpark, in order to compare the results.

4.1.1 MLxtend

In the context of A-Priori, MLxtend provides an easy-to-use implementation for association rule mining. The A-Priori function in MLxtend helps identify frequent itemsets from transaction data, while the association rules function generates association rules based on these frequent itemsets.

An initial grouping is performed between users and their reviewed books, creating the baskets — a set of items (books) that a user has reviewed. The goal is then to build a sparse matrix, where the rows represent users (or transactions) and the columns represent individual books. Each entry in the matrix indicates whether a user has interacted with a specific book (1 for interaction, 0 for no interaction). The sparse matrix format is memory-efficient, making it ideal for large datasets where most interactions are missing (i.e., not all users interact with all books).

The first step to run the algorithm is to choose a support threshold s. This threshold means that if a set of items appears in at least s% of the baskets, it is considered frequent. In this case, the support threshold is set to 1.2% because it was the maximum percentage at which meaningful results were still obtained.

The A-Priori algorithm is applied to the sparse matrix obtained from the dataset, effectively removing less relevant combinations and ensuring that the resulting itemsets reflect statistically significant patterns in user interactions across the dataset.

In Table 4, the top 15 most frequent itemsets are presented. The highest observed support is 0.01298 (equivalent to 710 users), confirming that setting a higher support threshold would not have yielded any results.

Itemsets	**Support**
(B000ILIJE0, B000NWU3I4)	0.012980
(B000ILIJE0, B000PC54NG)	0.012869
(B000PC54NG, B000NWU3I4)	0.012850
(B000ILIJE0, B000PC54NG, B000NWU3I4)	0.012850
(B000NWQXBA, B000ILIJE0)	0.012832
(B000NWQXBA, B000PC54NG)	0.012832
(B000NWQXBA, B000ILIJE0, B000PC54NG)	0.012832
(B000NWQXBA, B000NWU3I4)	0.012813
(B000NWQXBA, B000PC54NG, B000NWU3I4)	0.012813
(B000NWQXBA, B000ILIJE0, B000NWU3I4)	0.012813
(B000NWQXBA, B000ILIJE0, B000PC54NG, B000NWU3I4)	0.012813
(B000Q032UY, B000PC54NG, B000H9R1Q0)	0.012758
(B000NWQXBA, B000H9R1Q0)	0.012758
(B000Q032UY, B000PC54NG)	0.012758
(B000ILIJE0, B000Q032UY)	0.012758

Table 4: Frequent Itemsets and their Support

A previously said, the Table 4 results show that in itemsets with more than two elements, the pairs of items that appear together also appear as frequent pairs on their own. For example, the triplet in the fourth row (B000ILIJE0, B000PC54NG, B000NWU3I4) is formed by the union of the first three rows.

4.1.2 Custom Implementation

After implementing the algorithm using the built-in function, a manually implementation of the algorithm is needed to work in PySpark. Unlike the previous approach, in this case, only pairs of books are considered instead of larger itemsets, to reduce the complexity of processing and filtering frequent itemsets. While this approach may not capture more complex relationships between multiple books, it still provides valuable insights into commonly associated book pairs, which can be useful for recommendation system.

The final dataset in PySpark (Table 3) is transformed into an RDD (Resilient Distributed Dataset) by extracting relevant columns (`User_id` and `Id`) and grouping them by `User_id` and then, for each user, by creating a set of unique items (books) that they have interacted with.
The A-priori algorithms is implemented through the following steps:

- The algorithm receive as input the RDD of book lists and the chosen support threshold of 50;

- Create candidate itemsets by considering all possible pairs of books for each user and sorting them to ensure consistency;
- Map function: maps each book pair to $(pair, 1)$;
- Reduce function: reduces by key to count the number of occurencies for each pair;
- Filter the pairs by keeping only those that appear at least as many times as the given threshold;
- Return an RDD containing only the frequent pairs along with their corresponding counts.

Itemsets	**Count**
(B000ILIJE0, B000NWU3I4)	710
(B000NWU3I4, B000PC54NG)	704
(B000ILIJE0, B000PC54NG)	704
(B000NWQXBA, B000PC54NG)	702
(B000NWQXBA, B000NWU3I4)	702
(B000ILIJE0, B000NWQXBA)	702
(B000H9R1Q0, B000PC54NG)	696
(B000NWQXBA, B000Q032UY)	696
(B000ILIJE0, B000Q032UY)	696
(B000H9R1Q0, B000Q032UY)	696
(B000H9R1Q0, B000NWU3I4)	696
(B000H9R1Q0, B000ILIJE0)	696
(B000PC54NG, B000Q032UY)	696
(B000H9R1Q0, B000NWQXBA)	696
(B000NWU3I4, B000Q032UY)	696

Table 5: Top 15 Frequent Book Itemsets

The previous Table 5 shows the top 15 most frequent itemsets, confirming the correctness of the methodological approach, as the result matches the one obtained previously using the built-in A-Priori function.

To identify hidden relationships between items in a dataset, association rules are used (shown in Table 6). Key metrics such as support, confidence, and lift help evaluate the strength of these associations. The form of association rule is $I \rightarrow j$, where I is a set of items and j is an item. The implication is that if all of the items in I appear in some basket, then j is "likely" to appear in that basket as well.

Consider the first row of Table 6 as an example to explain the metrics that appear.

book_B	**book_A**	**pair_count**	**count_A**	**count_B**	**support**	**conf AtoB**	**conf BtoA**	**lift**
B000NWU3I4	B000ILIJE0	710	711	710	0.0131	0.9986	1.0	75.8186
B000PC54NG	B000NWU3I4	704	710	704	0.0130	0.9915	1.0	75.9253
B000PC54NG	B000ILIJE0	704	711	704	0.0130	0.9901	1.0	75.8185
B000NWU3I4	B000NWQXBA	702	702	710	0.0130	1.0	0.9887	75.9253
B000PC54NG	B000NWQXBA	702	702	704	0.0130	1.0	0.9971	76.5724
B000NWQXBA	B000ILIJE0	702	711	702	0.0130	0.9873	1.0	75.8185
B000PC54NG	B000H9R1Q0	696	696	704	0.0129	1.0	0.9886	76.5724
B000Q032UY	B000NWU3I4	696	710	696	0.0129	0.9802	1.0	75.9253
B000NWQXBA	B000H9R1Q0	696	696	702	0.0129	1.0	0.9914	76.7905
B000ILIJE0	B000H9R1Q0	696	696	711	0.0129	1.0	0.9789	75.8185
B000NWU3I4	B000H9R1Q0	696	696	710	0.0129	1.0	0.9802	75.9253
B000Q032UY	B000H9R1Q0	696	696	696	0.0129	1.0	1.0	77.4525
B000Q032UY	B000NWQXBA	696	702	696	0.0129	0.9914	1.0	76.7905
B000Q032UY	B000PC54NG	696	704	696	0.0129	0.9886	1.0	76.5724
B000Q032UY	B000ILIJE0	696	711	696	0.0129	0.9789	1.0	75.8185

Table 6: Association Rules

- **book_B, book_A**: indicates the identifier of the books that are part of a pair of items. The most frequent pair is composed by B000NWU3I4 and B000ILIJE0, which are the identifiers for books B and A, respectively;

- **pair_count**: the number of times the pair of books was rated together. B000NWU3I4 and B000ILIJE0 have been rated togheter 710 times;

- **count_B, count_A**: the number of times each book was rated, independently of the other. In this case B000NWU3I4 and B000ILIJE0 have been rated 711 and 710 times respectively;

- **support**: the relative frequency of the pair of books compared to the total number of transactions. The support for the considered pair of books is 0.0131, meaning that the pair of books is reviewed together approximately 1.31% of the time;

- **conf AtoB, conf BtoA**: indicates the probability that book_B (book_A) will be purchased given that book_A (book_B) has already been purchased. In this case, the confidence from A to B is 0.9986, meaning that if book A (B000ILIJE0) is bought, there is a 99.86% chance that book B (B000NWU3I4) will also be bought. Similarly, the confidence from B to A is 1.0, meaning that if book B (B000NWU3I4) is bought, there is a 100% chance that book A (B000ILIJE0) will also be bought;

- **lift**: measures how much more likely the pair of books book_A and book_B are to be purchased together compared to when the two books are purchased separately. A lift greater than 1 indicates that the two books are purchased together more often than expected by chance. The lift is 75.8186, which indicates that the pair of books is 75.82 times more likely to be purchased together than if the books were purchased independently.

4.2 SON Algorithm

The SON (Savasere, Omiecinski, and Navathe) algorithm is an efficient technique for mining frequent itemsets in large datasets, especially when the dataset is too large to fit into memory. The algorithm is designed to handle large datasets by dividing them into smaller chunks, which are processed in parallel to improve computational efficiency. When implemented using MapReduce, the SON algorithm runs in two phases: initially, it identifies locally frequent itemsets within each chunk, and then merges these to determine the global frequent itemsets across the entire dataset. This partitioning and two-phase approach helps reduce memory usage and enhances performance in distributed environments. By distributing the computation across multiple nodes, this approach significantly improves scalability, making it ideal for processing large datasets in parallel computing environments.

The dataset (containing user IDs and the corresponding baskets of reviewed items) is divided into 10 partitions, and the minimum support thresholds are set: $p = 5$ is the local support, computed as the ratio between the global support and number of partitions, and $s = 50$ is the global support and mantains coherence with the previously implemented algorithm.

In the first phase, locally frequent itemsets are identified within each chunk, and these itemsets are merged into a set of candidate itemsets.

- First Map Function: outputs key-value pairs (F, 1), with F indicating the frequent itemset from the sample.
- First Reduce Function: outputs the candidate itemsets -distinct itemsets that appear one or more times.

The output of the first phase can be seen in following Table 7.

In the second phase, the global support for these candidate itemsets is calculated across all chunks, and the final frequent itemsets are derived by filtering those that meet the global support threshold.

Itemset	Frequency
{B0009K75X6, B000GRZI8Q}	1
{B00017JJ5E, B000K0DB8I}	1
{0451513967, B000C1X8JC}	1
{9562910334, B000P3LVZA}	1
{B0006BV75A, B000EVFCRG}	1
{B000CRFW9A, B000OVMRLA}	1
{0141804459, B0006AQ4LI}	1
{B000IVDZR6}	1
{1578152437}	1
{B0001FZGSK}	1

Table 7: Candidate itemsets

- Second Map Function: take all the output from the first Reduce function. Each Map task counts the number of occurences of each itemsets in the considered chunk. Output: (C,v) where C is one candidate set and v is the support among the baskets that were input to the map.
- Second Reduce Function: adds up the associated values and the final result is the total support for each itemset. Those itemsets whose total support is at least s, are frequent in the whole dataset.

The output of the second phase, which contains SON Algorithm 15 most frequent itemsets, can be seen in the following Table 8.

Itemset	Frequency
{B000ILIJE0, B000NWU3I4}	710
{B000ILIJE0, B000PC54NG}	704
{B000NWU3I4, B000PC54NG}	704
{B000NWQXBA, B000PC54NG}	702
{B000NWQXBA, B000NWU3I4}	702
{B000ILIJE0, B000NWQXBA}	702
{B000PC54NG, B000Q032UY}	696
{B000NWQXBA, B000Q032UY}	696
{B000NWU3I4, B000Q032UY}	696
{B000H9R1Q0, B000ILIJE0}	696
{B000H9R1Q0, B000PC54NG}	696
{B000H9R1Q0, B000Q032UY}	696
{B000H9R1Q0, B000NWU3I4}	696
{B000H9R1Q0, B000NWQXBA}	696
{B000ILIJE0, B000Q032UY}	696

Table 8: Top 15 most frequent itemsets

4.3 Algorithms Results Comparison

As shown in the following Table 9 the results from both the A-Priori and SON algorithms are largely consistent, with identical frequent itemsets appearing in both methods. The itemsets are presented alongside their respective support counts, indicating the frequency with which these pairs of books were rated together.

A-priori Result	**Support**	**SON Result**	**Support**
('B000ILIJE0', 'B000NWU3I4')	710	('B000ILIJE0', 'B000NWU3I4')	710
('B000ILIJE0', 'B000PC54NG')	704	('B000NWU3I4', 'B000PC54NG')	704
('B000NWU3I4', 'B000PC54NG')	704	('B000ILIJE0', 'B000PC54NG')	704
('B000NWQXBA', 'B000PC54NG')	702	('B000NWQXBA', 'B000PC54NG')	702
('B000ILIJE0', 'B000NWQXBA')	702	('B000NWQXBA', 'B000NWU3I4')	702
('B000NWQXBA', 'B000NWU3I4')	702	('B000ILIJE0', 'B000NWQXBA')	702
('B000NWQXBA', 'B000Q032UY')	696	('B000H9R1Q0', 'B000PC54NG')	696
('B000H9R1Q0', 'B000PC54NG')	696	('B000NWQXBA', 'B000Q032UY')	696
('B000H9R1Q0', 'B000NWU3I4')	696	('B000ILIJE0', 'B000Q032UY')	696
('B000NWU3I4', 'B000Q032UY')	696	('B000H9R1Q0', 'B000Q032UY')	696
('B000H9R1Q0', 'B000NWQXBA')	696	('B000H9R1Q0', 'B000NWU3I4')	696
('B000H9R1Q0', 'B000ILIJE0')	696	('B000H9R1Q0', 'B000ILIJE0')	696
('B000PC54NG', 'B000Q032UY')	696	('B000PC54NG', 'B000Q032UY')	696
('B000ILIJE0', 'B000Q032UY')	696	('B000H9R1Q0', 'B000NWQXBA')	696
('B000H9R1Q0', 'B000Q032UY')	696	('B000NWU3I4', 'B000Q032UY')	696

Table 9: Comparison of Top 15 Frequent Itemsets (A-priori vs SON)

For example, the pair ('B000ILIJE0', 'B000NWU3I4') in the first row appears in both the A-priori and SON results with a support count of 710. Similarly, other pairs such as ('B000ILIJE0', 'B000PC54NG') and ('B000NWU3I4', 'B000PC54NG'), second and third rows respectively, also appear consistently in both methods, each with a support count of 704.

The overall agreement between the two algorithms suggests that both are effective in identifying the most frequent itemsets within the dataset, as the results align perfectly across both methods. This reinforces the validity of the findings and indicates that the applied methodologies are robust.

5 Expand on Larger Dataset

To demonstrate the scalability of the algorithms, we ran the A-priori algorithm on the original dataset after pre-processing. The dataset was first transformed once again into an RDD (Resilient Distributed Dataset) that was then passed to the function that implements the A-priori algorithm to find frequent item pairs, using a predefined support threshold.

Itemset	**Frequency**
{B000ILIJE0, B000NWU3I4}	3450
{B000ILIJE0, B000PC54NG}	3428
{B000NWQXBA, B000PC54NG}	3424
{B000ILIJE0, B000NWQXBA}	3424
{B000NWU3I4, B000PC54NG}	3423
{B000NWQXBA, B000NWU3I4}	3418
{B000NWQXBA, B000Q032UY}	3400
{B000ILIJE0, B000Q032UY}	3400
{B000PC54NG, B000Q032UY}	3400
{B000H9R1Q0, B000PC54NG}	3396
{B000H9R1Q0, B000Q032UY}	3396
{B000H9R1Q0, B000ILIJE0}	3396
{B000H9R1Q0, B000NWQXBA}	3396
{B000NWU3I4, B000Q032UY}	3394
{B000H9R1Q0, B000NWU3I4}	3390

Table 10: Top 15 Frequent Itemsets (Full Dataset)

The results in Table 10 reveal that the top 15 most frequent itemsets are the same as those observed in previous runs, confirming that the implemented algorithms are scalable. The ability to process the entire dataset without a significant drop in performance demonstrates the robustness and efficiency of the used approach.

6 Conclusion

In this project, market basket analysis is explored as a fundamental technique in data mining, highlighting its value in identifying associations between products. The implementation of algorithms for large-scale data allowed to detect recurring purchasing patterns, which can support strategic decision-making in various sectors, such as retail and marketing.

It should be noticed that this project contains some limitations due to the high computational cost associated with analyzing a dataset of enormous size. Despite using PySpark to parallelize the processes, certain restrictions are imposed, such as calculating only pairs instead of considering larger baskets. Thus, for future research, it would be interesting to apply more advanced algorithms and analyze all the data for each methodology in order to obtain even more accurate and adaptive predictions.