Market Basket Analysis

Sara Sartini, Arianna Girotto

27 October 2025

1 Data description and preparation

The dataset used in this project is **Amazon Books Reviews**, published on Kaggle, and it contains a large collection of reviews related to books sold on Amazon, along with columns such as product ID, book title, user identifier, rating, and review text.

The project aims to implement a system to find frequent itemsets (market-basket analysis): the detector can consider as baskets the strings contained in the `review/text` column, using words as items, or the set of books reviewed by the same user, using books as items.

For this reason, only the `review` table from the original dataset will be used, and within this table, only the following columns will be retained:

Column Name	Type	Description	Usage
`Title`	String	Title of the reviewed book	Each title is an item in a basket
`User_id`	Integer	Identifier of the user who wrote the review	Each `User_id` represents a user, and each user is a basket
`review/text`	String	Textual content of the review	Each word is an item in a basket

Table 1: Selected columns from the `review` table of the dataset.

Based on the two different settings of the project, the data were prepared in two different ways:

1. In the case where **book are items** and **users are baskets**:
   - First rows with missing values were dropped. Specifically the following missing were found:

Column	NaN
`Title`	208
`User_id`	561,787
`review/text`	8

Table 2: Number of missing values per column

     All rows with missing values were removed (with the exception of `Review/Text`, which is not used in this analysis): since each `User_id` uniquely identifies a basket, rows with missing `User_id` values were discarded, as it would not be possible to assign those books to any basket. After this filtering step, the dataset size decreased from 3,000,000 to 2,438,018 rows.
   - Then the `Title` column was normalized by removing spaces and punctuation, and converting all characters to lowercase.

Original Title	Normalized Title
Dr. Seuss: American Icon	drseussamericanicon
Its Only Art If Its Well Hung!	itsonlyartifitswellhung
Wonderful Worship in Smaller Churches	wonderfulworshipinsmallerchurches

Table 3: Examples of title normalization

2. In the case where **words are items** and **reviews are baskets**:

   - First, rows with missing values were dropped, and based on Table 2, only 8 rows were removed. Since in this setting each row corresponds to a basket and only the `review/text` column is required, there was no need to drop the rows with missing `User_id` values.
   - Then, the `review/text` column was preprocessed in order to transform each review into a set of tokens representing the items in the basket. Specifically, the text was converted to lowercase to ensure uniformity, and punctuation was removed to avoid treating the same words with different formatting as distinct tokens. The text was then split into individual words (tokenization), and common stopwords in English (e.g. "the", "and", "of") were removed, since they do not carry meaningful information for the analysis.

Original Review	Tokenized Review
Trams (or any public transport) are not usually the best place to read and absorb scholarly texts.	[trams, public, transport, usually, best, place, read, absorb, scholarly, texts]
I just finished the book, "Wonderful Worship in Smaller Churches"	[just, finished, book, wonderful, worship, smaller, churches]

Table 4: Examples of review tokenization

2 Multistage Algorithm (*Books as Items*)

The `multistage` procedure begins with the definition of *preliminary parameters* and *utility functions* that will be used throughout the algorithmic pipeline. This initial setup establishes how user baskets are constructed, how the support threshold is determined, and how candidate itemsets are efficiently filtered through hashing.

1. **Construction of Baskets.** Each `User_id` is associated with a set of book titles, forming what is commonly called a *basket*:

   $$\texttt{user\_baskets} = \big\{\, \{\texttt{Title}_1, \ldots, \texttt{Title}_m\} \text{ for each user} \,\big\}.$$

   Duplicated titles within the same basket are removed so that the support reflects the *presence* of an item per basket rather than its multiplicity: this aligns with the classical market-basket assumption that each user either has or does not have a given item.

2. **Support Threshold.** The support threshold is a key parameter for pruning the search space: only itemsets with a support count greater than or equal to s_{abs} are retained as frequent, while all others are discarded.

   Given a minimum *relative* support $\texttt{min\_support} \in (0, 1]$ and $N = |\texttt{user\_baskets}|$ total baskets, the corresponding *absolute* support threshold is computed as:

   $$s_{\text{abs}} = \max\big(1,\ \lfloor \texttt{min\_support} \times N \rfloor\big).$$

   where:

   - `min_support` expresses the minimum fraction of baskets in which an itemset must appear,
   - s_{abs} is the equivalent absolute frequency count in the dataset.

Example. If `min_support` $= 0.01$ and the dataset contains $N = 1{,}000{,}000$ baskets, then:

$$s_{\text{abs}} = 0.01 \times 1{,}000{,}000 = 10{,}000.$$

This means that an itemset must appear in at least 10,000 baskets to be considered frequent.

Choosing an appropriate support threshold is a crucial step in frequent itemset mining: if the minimum support is set too low, the algorithm will return an excessively large number of itemsets, while setting the threshold too high may cause relevant patterns to be missed.

To explore this trade-off, a grid of minimum support values has been evaluated:

$$\texttt{min\_support} \in \{0.0005,\ 0.001,\ 0.002,\ 0.005,\ 0.01,\ 0.02,\ 0.05\}.$$

3. **Utility Functions for Hashing.** To reduce the number of candidate itemsets that will be counted in later stages, the algorithm applies a two-stage deterministic hashing scheme:

   - `tuple_key(items)` produces a canonical representation of an itemset by sorting its elements, concatenating them with the separator `"||"`, and encoding the resulting string in UTF-8: this ensures that permutations of the same itemset map to the same key.
   - `stable_bucket(tuple_key, num_buckets, seed)` applies the `blake2b` cryptographic hash function with a custom *personalization string* based on `seed`, and maps the key to an integer bucket in the range $[0, \texttt{num\_buckets})$.

   This strategy ensures both *determinism* and *independence* between hashing stages, enabling efficient filtering of infrequent candidates before the costly support counting step.

The procedure iterates on $k = 1, 2, \ldots, k_{\max}$ and returns all frequent itemsets up to size $k_{\max}$.

1. **Frequent 1-itemsets (L_1).** The first step consists in identifying which individual items are frequent: a single scan of the dataset counts the occurrence of each item across all baskets and items whose support is greater than or equal to the absolute threshold s_{abs} are retained as frequent 1-itemsets:

   $$L_1 = \big\{\{i\} \ :\ \text{support}(i) \geq s_{\text{abs}}\big\}.$$

   Each itemset is stored as a `frozenset` object.

   Suppose that there are 4 baskets and they contains the following list of books:

Basket (User)	**Titles**
B_1	{*drseussamericanicon*, *itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*}
B_2	{*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*, *dramaticaforscreenwriters*}
B_3	{*wonderfulworshipinsmallerchurches*, *whispersofthewickedsaints*}
B_4	{*drseussamericanicon*, *wonderfulworshipinsmallerchurches*}

   A single pass through these baskets yields the following support counts:

Item (Book Title)	**Support Count**
drseussamericanicon	2
itsonlyartifitswellhung	2
wonderfulworshipinsmallerchurches	4
whispersofthewickedsaints	1
dramaticaforscreenwriters	1

   If we set $s_{\text{abs}} = 2$, the frequent 1-itemsets are:

   $$L_1 = \big\{\{\textit{itsonlyartifitswellhung}\},\ \{\textit{wonderfulworshipinsmallerchurches}\},\ \{\textit{drseussamericanicon}\}\big\}.$$

   Notice that items appearing in fewer than s_{abs} baskets (*whispersofthewickedsaints* and *dramaticaforscreenwriters*) are pruned at this stage.

2. **First hashing stage (bitmap$_1$).** In this step, each candidate k-itemset is mapped to a bucket using a deterministic hash function: for each basket, only the items that appear in L_1 are considered (*early filtering*), which avoids generating combinations containing infrequent items. For every k-combination $\binom{|B_u|}{k}$ present in the filtered basket, the algorithm computes the canonical key of the combination, hashes it into a bucket of the first hash table of size B_1 and increments the corresponding bucket count.

   After scanning all baskets, a bitmap is created:

$$\text{bitmap}_1[b] = \mathbb{1}\{\text{bucket\_count}_1[b] \geq s_{\text{abs}}\},$$

   where $\mathbb{1}\{\cdot\}$ denotes the indicator function; buckets whose counts are below the support threshold are set to zero and any candidate mapping to these buckets is immediately discarded.

   Now let's consider the same baskets introduced previously, supposing the analysis is about $k = 2$ (pairs of items). The frequent 1-itemsets are:

$$L_1 = \big\{\{\textit{itsonlyartifitswellhung}\},\ \{\textit{wonderfulworshipinsmallerchurches}\},\ \{\textit{drseussamericanicon}\}\big\}.$$

   For each basket, all 2-combinations *only* using these frequent items are generated:

Basket	**2-combinations from L_1**
B_1	(*drseussamericanicon*, *itsonlyartifitswellhung*) (*drseussamericanicon*, *wonderfulworshipinsmallerchurches*) (*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*)
B_2	(*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*)
B_3	(*wonderfulworshipinsmallerchurches*) (no pair, only 1 frequent item)
B_4	(*drseussamericanicon*, *wonderfulworshipinsmallerchurches*)

   Each 2-combination is then hashed into the first hash table and the counts for each bucket depend on the specific hash function, but conceptually the process can be summarized as:

2-itemset (canonical key)	**Bucket Count (after hashing)**
(*drseussamericanicon*, *itsonlyartifitswellhung*)	1
(*drseussamericanicon*, *wonderfulworshipinsmallerchurches*)	2
(*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*)	2

   With $s_{\text{abs}} = 2$, only the second and third buckets remain set in bitmap$_1$:

$$\text{bitmap}_1 = \{\text{1st bucket: } 0,\ \text{2nd bucket: } 1,\ \text{3rd bucket: } 1\}.$$

   This means that the pair (*drseussamericanicon*, *itsonlyartifitswellhung*) is discarded after the first hashing stage, since its bucket count is below the threshold.

3. **Second hashing stage (bitmap$_2$).** Candidates that survive the first filtering step are re-hashed using an *independent* hash function, which maps each candidate into a bucket of a second hash table of size B_2. The procedure mirrors the first hashing stage, so for each occurrence of a candidate in the baskets, its bucket count is incremented.

   Once all baskets have been processed, a second bitmap is constructed:

$$\text{bitmap}_2[b] = \mathbb{1}\{\text{bucket\_count}_2[b] \geq s_{\text{abs}}\}.$$

   Only candidates that map to buckets above the threshold in *both* bitmap$_1$ and bitmap$_2$ proceed to the final support counting step.

   From the first hashing stage, only the following 2-itemsets survived:

$$(\textit{drseussamericanicon}, \textit{wonderfulworshipinsmallerchurches})$$
$$(\textit{itsonlyartifitswellhung}, \textit{wonderfulworshipinsmallerchurches})$$

   These are now re-hashed into the second table of size B_2; after scanning the baskets again, the bucket counts are conceptually as follows:

2-itemset (canonical key)	Bucket Count (after second hashing)
(*drseussamericanicon*, *wonderfulworshipinsmallerchurches*)	2
(*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*)	2

Since both counts are $\geq s_{\text{abs}} = 2$, their corresponding bits in bitmap_2 are set to 1.

$$\text{bitmap}_2 = \{\text{1st surviving bucket: } 1,\ \text{2nd surviving bucket: } 1\}.$$

At this point, both candidate pairs have passed the double hashing filter and will be evaluated in the *true support counting* step: any candidate failing this second filter would have been discarded without further computation.

4. **True support counting and pruning (L_k).** In the final counting step, the algorithm scans the baskets once more and counts the *true support* for all candidates that survived both hashing stages: this ensures that the final set of frequent itemsets is exact, despite the use of probabilistic filters in the previous steps.

   All candidates with support count greater than or equal to the absolute threshold s_{abs} are retained as frequent k-itemsets:

   $$L_k = \big\{X \in C_k : \text{support}(X) \geq s_{\text{abs}}\big\}.$$

   After the two hash filtering stages, the two 2-itemsets that passed both are:

   (*drseussamericanicon*, *wonderfulworshipinsmallerchurches*)
   (*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*)

   Their true support is counted across the 4 baskets:

2-itemset	True Support Count
(*drseussamericanicon*, *wonderfulworshipinsmallerchurches*)	2
(*itsonlyartifitswellhung*, *wonderfulworshipinsmallerchurches*)	2

   Since both counts are $\geq s_{\text{abs}} = 2$, they are retained in L_2:

   $$L_2 = \Big\{ \begin{array}{l} \{\textit{drseussamericanicon}, \textit{wonderfulworshipinsmallerchurches}\}, \\ \{\textit{itsonlyartifitswellhung}, \textit{wonderfulworshipinsmallerchurches}\} \end{array} \Big\}.$$

   If no candidate had reached the threshold, L_2 would have been empty and the algorithm would have stopped at this point.

5. **Iteration and termination.** Once L_k has been determined, the algorithm increments the size of the itemsets ($k \leftarrow k + 1$) and repeats Steps 2–5, using L_k to generate new candidates for the next level. This iterative process continues until either:

   - no new frequent itemsets are found ($L_k = \varnothing$), or
   - the predefined maximum size $k_{\max}$ is reached.

   The final output is the union of all frequent itemsets identified at each iteration:

   $$L = \bigcup_{k=1}^{k_{\max}} L_k,$$

   with each itemset reported together with its absolute count and relative support (count/N).

2.1 Experimental results

The algorithm was executed on a dataset containing 1,008,961 user baskets, using the grid of minimum support values previously described.
The absolute support threshold s_{abs} was computed for each value of `min_support` and the number of frequent itemsets was recorded for $k = 2, 3, 4$.

The results are summarized in Table 2.1, which reports the number of frequent itemsets found for each support level and for $k = 2, 3, 4$.

min_support	k=2	k=3	k=4
0.0005	315	0	0
0.0010	78	0	0
0.0020	7	0	0
0.0050	0	0	0
0.0100	0	0	0
0.0200	0	0	0
0.0500	0	0	0

At very low support values (0.0005 and 0.001), the algorithm identifies a relatively large number of frequent 2-itemsets (315 and 78, respectively), and as the support threshold increases, the number of frequent patterns drops sharply, reaching zero at `min_support` = 0.005.
No frequent itemsets of size 3 or 4 were found for any threshold in this experiment, which suggests that co-occurrence beyond pairs of books is rare in this dataset.

3 The Savasere, Omiecinski and Navathe Algorithm (Words as Items)

The `SON` algorithm implements a two-phase frequent itemset mining strategy designed for distributed environments using `Apache Spark`: it combines the classic Apriori algorithm within each data partition with the SON framework, ensuring scalability and correctness under distributed computation.

The pipeline begins with the definition of *control parameters* and *pre-processing steps*:

1. **Construction of Baskets.** Each review is represented as a set of unique tokens:

   $$\texttt{baskets} = \big\{\, \{\texttt{token}_1, \ldots, \texttt{token}_m\} \text{ for each review} \,\big\}.$$

   Duplicate tokens within the same basket are removed to ensure that the support of each token reflects its presence in a basket rather than its frequency within a single document.

   To reduce computational load during experimentation, the dataset may optionally be subsampled using a predefined fraction $f \in (0, 1]$, controlled by the parameter:

   $$\texttt{SUBSAMPLE\_FRACTION} = f.$$

   this allows controlled scaling of the input size while preserving statistical properties of the data.

2. **Support Threshold.** Let $N = |\texttt{baskets}|$ denote the total number of baskets and $\texttt{RELATIVE\_SUPPORT} \in (0, 1]$ the chosen minimum relative support. The corresponding absolute support threshold is computed as:

   $$s_{\text{glob}} = \max\big(1,\ \lfloor \texttt{RELATIVE\_SUPPORT} \times N \rfloor\big),$$

   where s_{glob} specifies the minimum number of baskets in which an itemset must appear to be considered globally frequent.

   NB: Why using a subsample? The SON algorithm has been implemented in PySpark, which makes it inherently scalable with respect to dataset size. However, due to the memory constraints of the execution environment (e.g. *Google Colab* provides approximately 12 GB of RAM), the experiments were carried out on a representative 15% subsample of the original dataset (∼450,000 rows): subsampling was a practical choice to stay within hardware limits while preserving the algorithm's structure, performance, and scalability.

3. **SON Local Threshold Scaling.** The SON algorithm relies on the property that any globally frequent itemset must also be frequent within at least one partition relative to its local size.

   Let n_p be the number of baskets in a given partition p, and let

   $$p = \frac{n_p}{N}$$

   Then the local support threshold for partition p is:

   $$s_{\text{loc}}^{(p)} = \max\big(1,\ \lceil s_{\text{glob}} \times p \rceil\big).$$

Now the algorithm follows the following steps:

1. **Local phase.** To reduce both computation and communication costs, the Apriori algorithm is applied *locally* on each data partition to identify a set of "promising" global candidates, following the SON scheme:

   - **Local 1-itemsets.** Item frequencies are counted within the partition using a `Counter`, and those with counts $\geq s_{\text{loc}}$ are retained:

   $$L_1^{(P)} \;=\; \big\{\{i\} \;:\; \text{count}_P(i) \geq s_{\text{loc}}\big\}.$$

   - **Candidate generation for $k \geq 2$.** For $k = 2, 3, \ldots,$ `max_k`, candidate itemsets are generated through a self-join of the frequent itemsets from the previous iteration:

   $$C_k^{(P)} \;=\; \big\{\, I \cup J \;:\; I, J \in L_{k-1}^{(P)},\; |I \cup J| = k \,\big\}.$$

   Each candidate is counted in the baskets of the current partition, and only those with support $\geq s_{\text{loc}}$ are retained to form $L_k^{(P)}$.

   - **Emission of candidates.** The union of all local frequent itemsets up to size `max_k`,

   $$\bigcup_{k \leq \texttt{max\_k}} L_k^{(P)},$$

   is emitted as the set of local candidates from this partition.

2. **Union of global candidates.** All candidates produced by the partitions are then merged:

   $$\mathcal{C} \;=\; \bigcup_{\text{partitions } P} \; \bigcup_{k \leq \texttt{max\_k}} L_k^{(P)},$$

   where:

   - for each partition P, we take all the locally frequent itemsets (of any size up to `max_k`);
   - we put them all together into a single global "bag";
   - if the same itemset appears in multiple partitions, we keep it only once (no duplicates);
   - this gives us $\mathcal{C}$, the final list of global candidates.

3. **Global counting and final filtering.** In the second pass, candidates $\mathcal{C}$ are counted across *all* baskets and the absolute support threshold s_{abs} is applied; this step performs the *global scan* that confirms (or rejects) the frequency of itemsets proposed during the local phase:

   - `flatMap` For each basket, the algorithm checks all candidates $X \in \mathcal{C}$:
     - if X is contained in the basket, it emits the pair $(X, 1)$;
     - otherwise, it emits nothing.

     In this way, the output of this step contains one pair $(X, 1)$ for each actual occurrence of X in the entire dataset.
   - `reduceByKey` All the pairs $(X, 1)$ corresponding to the same candidate X are aggregated (summed) to obtain

     $$(X, \text{support}(X)).$$

     At this point, we know exactly how many baskets contain each candidate X.
   - `filter` Finally, the algorithm keeps only those candidates whose support is greater than or equal to the global support threshold s_{abs}:

     $$L \;=\; \big\{X \in \mathcal{C} : \text{support}(X) \geq s_{\text{abs}}\big\}.$$

     These are the true *globally frequent itemsets.*

3.1 Experimental results

The experiments were carried out with the following configuration:

Parameter	Value	Description
`USE_SUBSAMPLE`	`True`	Enables subsampling of the dataset to reduce memory usage.
`SUBSAMPLE_FRACTION`	0.15	15% random sample of the original dataset (∼450,000 rows).
`RELATIVE_SUPPORT`	0.1	Minimum relative support threshold (10%) for itemsets to be considered frequent.
`MAX_K`	3	Maximum size of frequent itemsets (1-, 2-, and 3-itemsets).

The algorithm produced frequent itemsets up to size $k = 3$ and the most frequent patterns by support where the following.

Itemset	Occurrences	Support (%)
{*book*, *read*, *one*}	73,256	16.28
{*book*, *like*, *read*}	53,240	11.83
{*it*, *read*, *book*}	49,061	10.90
{*book*, *would*, *read*}	49,050	10.90
{*story*, *book*, *read*}	47,647	10.59
{*books*, *read*, *book*}	47,207	10.49
{*book*, *like*, *one*}	46,523	10.34
{*book*, *time*, *read*}	46,217	10.27
{*book*, *reading*, *read*}	45,926	10.21

Table 5: Top frequent **3-itemsets** with absolute occurrences and relative support.

Itemset	Occurrences	Support (%)
{*book*, *read*}	166,445	36.99
{*book*, *one*}	126,692	28.15
{*book*, *like*}	93,253	20.72
{*read*, *one*}	88,176	19.59
{*book*, *would*}	86,831	19.30
{*story*, *book*}	79,266	17.61
{*good*, *book*}	79,129	17.58
{*book*, *it*}	77,661	17.26
{*book*, *great*}	77,052	17.12
{*book*, *time*}	76,910	17.09

Table 6: Top frequent **2-itemsets** with absolute occurrences and relative support.

Item	Occurrences	Support (%)
book	336,520	74.78
read	205,383	45.64
one	162,744	36.17
like	113,671	25.26
story	106,033	23.56
would	103,885	23.09
good	97,629	21.70
great	96,730	21.50

Item	Occurrences	Support (%)
time	96,038	21.34
books	94,534	21.01

Table 7: Top frequent **1-itemsets** with absolute occurrences and relative support.

The results confirm a strong dominance of the token *book* across all k.
Most 2- and 3-itemsets are structured around this word, reflecting the domain of the corpus (book reviews): co-occurrences such as {*book*, *read*} or {*book*, *read*, *one*} capture recurring linguistic patterns like "I would read this book" or "one book to read".

The report must also contain the following declaration: "I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work, and including any code produced using generative AI systems. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study