Università degli Studi di Milano

UNIVERSITÀ
DEGLI STUDI
DI MILANO

Market-basket Analysis:

Internet Movie Database (IMDb)

Algorithms for Massive Datasets (AMD)

Student
Francesco Lazzara

Degree Course
Data Science and Economics

ID
942830

I declare that this material, which I now submit for assessment, is entirely my 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 work. I understand that plagiarism, collusion, and copying are grave and serious offenses 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 or any other person for assessment on this or any other course of study.

Contents

1 Introduction

The aim of the research presented throughout this project is to implement a system to find frequent itemsets of different cardinality. The main source of data considered comes from the *IMDb* non-commercial dataset, which is composed by three files with two key fields used to perform join operations between the tables to reconstruct the useful associations. The task of the project consists in writing from scratch the phases of an algorithm presented during in the course of *Algorithms for Massive Datasets*, by considering "actors" as items and "movies" as baskets. In this setting, the framework used to develop and test the whole code was provided by *Google Colab* and consisted in a free hosted runtime that exploits Google Drive personal accounts to store the IMDb dataset, downloaded after a connection with the Kaggle API is established by using my personal tokens and uploading them on the notebook. Of course, among the different requirements set out for the task, the main code of the project should be developed by taking into account the main memory available in the virtual machine and the running time of each chunk of code, so that it could be properly replicated on other environments and scale well on larger amounts of data.

In the first section of the paper, we would briefly mention the essential steps to be followed in order to properly setting the environment and connect with the Kaggle API to correctly import the three tables, and we would shortly describe each feature. Moreover, we would perform some exploratory data analysis and data pre-processing tasks, aimed at building the final structure of the baskets file that would be used as input data of our algorithm. On the contrary, in the third chapter we would explain the theory behind the market-basket analysis tasks and the implementation of the widely known *A-Priori algorithm*, by giving reasoning about the passes it performs over the data and describing its main characteristics and differences with respect to an alternative class of algorithms. Moreover, we would say something about the assumptions that must hold in order to be able to scale-up the frequent itemsets system to larger datasets, aka bigger basket files, and how to properly set up the support threshold to achieve meaningful and interesting results. Then, the fourth chapter would cover a detailed description of each pass of the algorithm over the data, by subsequently illustrating each chunk of code written in the Python notebook, which corresponds to an iterative procedure, and would give some rational arguments to justify the criterion used to discriminate between frequent and non-frequent itemsets.

Subsequently, in chapter five we would inspect the ranking of the discovered frequent itemsets of different size, which top ten of most frequent itemsets is contained in the appendix section, and we would try to provide some evidences that validate our findings by relying on the additional information found on the *Wikipedia* and *IMDb* websites. The summary code containing the data pre-processing step and all the stages of the implemented algorithm, alongside with short comments, is uploaded in the publicly available Github folder of the project[1]. Accordingly, for a deeper understanding of the code and the tasks implemented throughout the analysis it is suggested to read this report and follow the procedure in the main code, where every stage and the corresponding results are interpreted and contextualized.

[1] Github folder of the project

2 Description of the Dataset

The dataset used throughout the analysis is taken from Kaggle[2] and it is imported and downloaded in Google Colab from the *Kaggle API*. The connection with the API is established by means of a token, downloaded as a JSON file, containing my API credentials (username and associated key) required to import the dataset remotely. Since the notebook is trusted under my Google account, which is connected to Colab, the first time the code is executed using a different account an authorization code is needed to mount the storage and download the dataset correctly. Hence, it would be asked to log-in with the google credentials and give permission to connect the Google Drive storage to Colab by copy-pasting an authorization code directly in the corresponding chunk code of Colab. The IMDb dataset, published under IMDb non-commercial licensing, was firstly created to make a movie/tv series recommendation system, while we would use to implement a system finding frequent itemsets. Therefore, once the JSON file is uploaded and *Spark* is installed, then we would assign the tables to three DataFrames: *titles*, *job* and *actors*.

2.1 Exploratory Data Analysis

In each of the three tables, we would filter the attributes by selecting the ones suitable for our task and needed to build up the final baskets file. The selected attributes are list below in combination with a short description of their measurements:

- **titles**.*tconst*: string; alphanumeric unique identifier of the title.
- **titles**.*titleType*: string; the type/format of the title (e.g. movie, short, tv-series, tv-episode, video, etc).
- **titles**.*PrimaryTitle*: string; the more popular title / the title used by the filmmakers on promotional materials at the point of release.
- **titles**.*originalTitle*: string; original title in the original language.
- **titles**.*genres*: string; includes up to three genres associated with the title.

- **job**.*tconst*: string; alphanumeric unique identifier of the title.
- **job**.*ordering*: integer; a number to uniquely identify rows for a given *tconst* (title id).
- **job**.*nconst*: string; alphanumeric unique identifier of the name/person.
- **job**.*category*: string; the category of job that person was in.

- **actors**.*nconst*: string; alphanumeric unique identifier of the name/person.
- **actors**.*primaryName*: string; name by which the person is most often credited.
- **actors**.*birthYear*: string; in YYYY format.
- **actors**.*deathYear*: string; in YYYY format if applicable else "Nan".
- **actors**.*knownForTitles*: string; titles the person is known for.

[2] IMDb Dataset

Once we filter the attributes and the null values (Nan) are removed, the three working datasets finally contain $6.321.295$, $36.468.817$ and $9.706.915$ observations, respectively. The "job" table is essential to perform join operations between "titles" and "actors" to create the baskets by means of two keys: *tconst* for the movies and *nconst* for the actors. Since our task is focused on considering as baskets the movies and as items the corresponding actors, we inspected from the "job" table the number of actors (actor/actress) in the dataset for any type of production (movies, tv-series, etc.), which is almost 15 millions, while among the "titles" DataFrame we counted 536.032 number of distinct movies.

Coherently, we constructed two tables, one with all the actors and the other with movies only, to match each movie with the corresponding actors, then we grouped the observations by their unique identifier ("tconst"), so that we do not combine actors from different movies in the same basket because of the title of the movie. Next, we retrieved the number of actors for each screenplay and we inspected the maximum number of actors contained in each basket overall. In particular, Table 1 shows the title of movies with the highest number of actors and we see that they all have 10 items, which is coherent with the assumption that the size of each basket is not that big and always fits main memory, while it is the size of the whole baskets file which could create some memory issues. On the contrary, if we just grouped the combined table by "titles" we get that there are some movies with the same names, given that now *Hamlet*, *Macbeth* and *Honeymoon* are the top three movies for the number of actors with 145, 140 and 139 characters, respectively. Those values are actually very far from the average number of actors per titles overall the baskets file, which is equal to $\mu = 4.3 \approx 4$ items.

Movies	**Number of actors**
The Grue Crew	10
The Gypsy Ball	10
Emmanuelle Through Time: Rod Steele 0014 & Naked Agent 0069	10
En Thangai Kalyani	10
The Break-In	10
Batman Beyond: Rising Knight	10
Stick To The Status Quo	10
Home in Time for War	10
El sexenio de la muerte	10
Another 'Brief' Case	10

Table 1: Top 10 movies with most number of actors

2.2 Data Pre-processing

At this stage, we would proceed by manipulating a bit the data so that they are ready to be given as input of our frequent itemsets system. Therefore, once we have grouped the observations by primary title, we would collect in a list all the actors belonging to each movie. In other words, we would represent the entire set of a baskets as one list that contains other lists, which is the representation of the baskets (title of the movies), while their elements correspond to the items (name of the actors). An alternative representation of the baskets file is through dictionary, where the tiles of the movies are the keys and the corresponding list of actors are the values. However, we would prefer to work with lists as they occupy less space in main memory since dictionaries need to store a pair of object for each basket, namely the title of the movies and the list of actors, thus it is more efficient to use lists without the needs of movie titles. In any case, the analysis would be carried out using a baskets file with cardinality equal to 393.653.

Subsequently, given that strings have more memory footprint rather than integers, we would prefer representing the items in baskets by converting the name of the actors to a unique number. Hence, we would initially retrieve the distinct actors by inspecting the entire set of baskets, then we would assign a unique integer in the interval $[0, 393.652]$, mainly because Python starts counting objects from zero. Accordingly, we would build sort of a hashing table by creating a dictionary of length equal to the number of distinct items, where the name the actors are used as keys and the integers to which they are mapped as values. At this point, we would convert the items of each basket in the entire set to their corresponding hashed values, by mapping each actor and retrieving from the dictionary the hashed integer value.

3 The A-Priori Algorithm and its Implementation

In *Market-basket Analysis* we aim to describe a many-to-many relationship between two objects: *baskets* and *items*. Commonly, each basket contains a set of items (itemset) with small cardinality, compared to the whole number of items, and we assume that each of them regularly fits the main memory, while on aggregate the overall set of baskets could potentially be too large to be contained in the RAM. For instance, if we have n distinct items, then there could be $\binom{n}{2} \approx \frac{n^2}{2}$ possible pairs, so that if we set $n = 10^5$ then we would need approximately $\frac{10^{10}}{2} \cdot 4_{bytes} \approx 20_{Gbytes}$ of RAM to store them all in main memory. The problem of discovery of *frequent* itemsets is mainly focused on getting the absolute number of baskets that contain a particular set of items with different size starting from a singleton, given that the empty set does not tell any useful information, and then by inspecting doubleton, triplets or quartets, etc. Simply, a set of items is said to be frequent when it appears in many baskets. In mathematical terms, this is equivalent of saying that given a support threshold s, a set of items I and the support of I $supp(I)$, equal to the number of baskets containing I as a subset, we would define a frequent itemset if $supp(I) > s$.

In this setting, the *A-Priori* algorithm works by eliminating from the large set of candidate itemsets the ones that cannot be frequent, starting from smaller sets to bigger ones. The main idea behind the A-Priori algorithm is that a set cannot be frequent unless the item(s) contained in its immediate subset(s) are not frequent themselves. For instance, suppose we have a triple composed by two out of three frequent items, then it would be meaningless to inspect whether the whole triple is frequent or not since it is essential that each pair of elements in the set must be a frequent doubleton. Of course, this criterion is valid for any itemsets of higher cardinality, thus if we end up finding only one frequent triple in all the baskets, then we are sure that is not possible to have a frequent quartets or larger sets. This fact is related to a factual conclusion defined as *Monotonicity Property* of the itemsets, which basically states what we have implicitly derived in the observation we made few lines before. In summary, this property ensures that if a set of items I is frequent, then every subset J that can be obtained from it is also frequent $J \subseteq I$, thus $supp(J) \geq supp(I)$. Therefore, once we set a specific support threshold s, then we define a *maximal* itemset when there are no frequent superset, equally it corresponds to the size of the largest frequent subset because no set that is a subset of the maximal itemset can be frequent.

In most of the applications, we are typically searching for small frequent sets, on the basis of their number of elements, so that if we indicate with k the size of the frequent itemsets we are interested in, then it would always lie in the finite interval $k \in [1, 3]$. However, if for any reasons we would need frequent itemsets of larger k-size, then with good chance as k gets higher the number of frequent elements among the whole set of the n-distinct ones falls, so that most of the elements in each basket might be eliminated as they are not candidate anymore. For those reasons, the A-Priori algorithm can be defined as an *exact* algorithm, in contrast to alternative systems or improvements described as *approximate* algorithms. The main differences between the two categories is that the latter typically have less running time, namely those algorithms work faster, but the final set of frequent items is not guaranteed to contain all of them, so some might still be missed or it might include some false positive/false negative. On the contrary, all the algorithms, irrespectively of their types (exact or approximate), are required to read the baskets sequentially, thus the fundamental feature that differentiates them lies in the number of times they pass through the basket file, which then defines their corresponding running time. Moreover, when dealing with massive datasets, it might be the case that the file containing the whole set of baskets does not fit in main memory, so the main cost of any algorithms is the time to read each basket from disk, which increases as the size of the frequent itemsets we are finding gets higher. In other words, we could say that the time an algorithm takes to pass over each basket and count the occurrences of each element is proportional to the actual size of the file.

The A-Priori algorithm consists of two main steps that are repeated every time the items in the baskets are read. In the first one, once we have converted the items of the baskets to integers, we perform a map and reduce task to compute the occurrences of each itemsets and then use the hashed values as keys to index the array of counts. Then, before moving to the second pass, we inspect the counts of the items and we determine, on the basis of a well-

specified support threshold s, which of them are candidate elements to be frequent as singletons and how many were actually excluded. Coherently, we would track the resulting state of the elements by using a *Frequent-item Table*, that in the case of singletons is implemented as an array of length n, where each item i is assigned to the i-th position (index equal to the hashed value) and the entries commonly correspond to 0 if item i is non-frequent or to the corresponding hashed integer in the opposite case.

Now, in the second pass of the A-Priori algorithm we focus on counting all the pairs, or itemsets of size $k = 2$, that consist of two previously found frequent items. Moreover, thanks to the monotonicity property, introduced before, we are sure we would not miss any frequent pairs and, depending on the value of the threshold s, the space of main memory required to read the data again would be less since all the non-frequent elements are not part of this stage anymore since they were removed. Consequently, the second pass works by mapping each basket and looking for all pairs of frequent items, then we would reduce them and add one to their corresponding support values every time a particular pair is found in any baskets. After all, the absolute counts of each pairs can be inspected and compared with the support threshold s to determine the actual amount of frequent itemset left, with this procedure that can be repeated for further analysis if frequent itemset of larger size k are useful to be inspected. Therefore, we would initially execute the chunks of code corresponding to each pass of the A-Priori algorithm up to the frequent pairs, while to inspect the frequent itemsets of larger size $k \in [3, 4]$ we would rely on the constructed `APriori()` function and we would validate the results with the ones obtained from the manual procedure.

3.1 Scaling of the Proposed Solution

The pattern of the two main passes of the A-Priori algorithm that is described in the lines before can be repeated as much as we want, preferably in the case we have a particular interest for frequent itemsets of larger size $k' > k$, where we usually define k in the interval $k \in [1, 3]$ which is the commonly inspected cardinality of elements in a frequent set. In this context, a key role is given to the pre-defined support threshold s, which highly depends on the data domain and is set on the basis of the expected results we would like to achieve. In other words, to give meaning to a market-basket analysis we need the resulting frequent itemsets to be a "significantly small" relative percentage with respect to the distinct combinations of elements that can appear together in a set, namely we the ones which truly occur the most. Coherently, since we assume that the baskets file is massive and could potentially not fit main memory, then to be able to perform our task it is essential that after each pass of the A-priori algorithm we are left with not too many frequent itemsets of different size k to be stored in the data structure for the counts.

In this circumstances, the assumption of *Sparsity of frequent itemsets* is fulfilled in practice when the support threshold s is set high enough that we end up only with "rare" frequent sets, which can be stored in main memory table. Therefore, at the end of each pass of the A-Priori algorithm the number of frequent sets would decrease as their size k increases, so that there would not be too many frequent pairs, triples, etc. to be read at each step, otherwise it would be impossible to count every combinations of itemsets for any size k. This is the main idea behind the workflow of every algorithms suitable for these kind of tasks, including the A-Priori algorithm and its variations, that is shortly to avoid counting an enormous quantity of candidate sets. In summary, if the assumption of sparsity of frequent itemsets holds, then the solution implemented in the main code of this project can scale well to bigger datasets (baskets file) providing that the support threshold s is set properly so to have at each step a convenient amount of candidates itemsets.

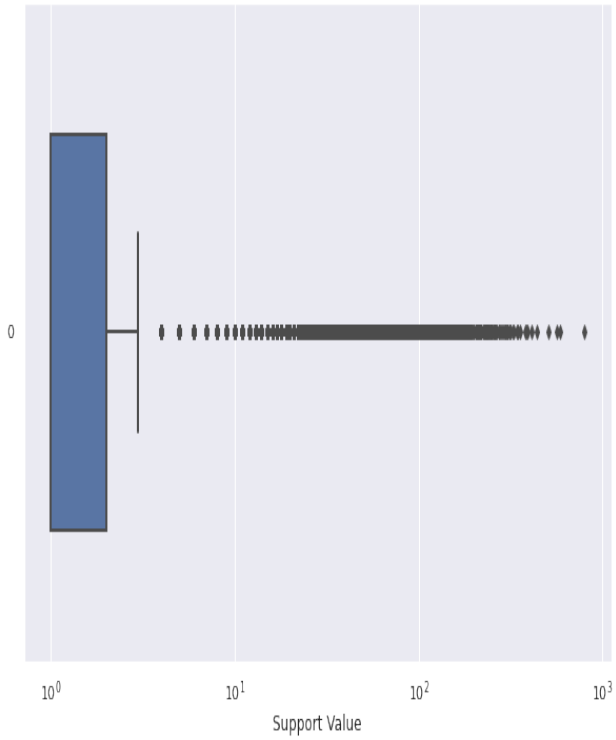
4 Description of the Experiments

As introduced in the previous section, all the algorithms that are suitable to find frequent itemsets necessarily need to pass multiple times over all the baskets and need to maintain several counts for the occurrences of single or multiple combination of items contained in whole baskets file. Therefore, it is crucial to have sufficient main memory to store the counts and update them faster than reading all the baskets from disk, which is considerably slower, even though resources are not unlimited and not all the items can be counted if they exceed the maximum available space. In fact, there are some approaches that can be implemented from the beginning of the analysis to efficiently represent items in baskets, when it is of course meaningful. Accordingly, as we already performed in the pre-processing phase, we have converted the items in each basket from character strings to positive integers with the aid of an hash table, which stores the name of each distinct actors associated to consecutive integers in the interval $[1, n]$. Hence, once the items are mapped to their corresponding value in the hash table, then there would be a greater space in main memory with good chance, as integers have less memory footprint than strings. Coherently, we represented the whole file as a big list having as elements as many lists as the total number of baskets, each one with items formatted as integers.

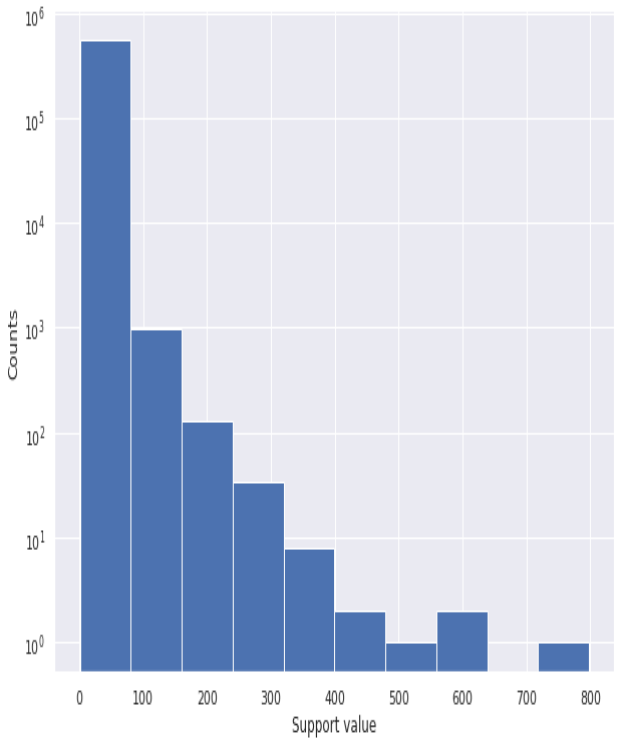
At this stage, the first pass of the A-Priori algorithm can start, coherently with all the steps described in the third chapter, by parallelizing the list of baskets to make the computations ready to be distributed among multiple processors. Hence, we would performa a *MapReduce* task where each item in the baskets would be used for the indexing of the hash table, which would then be reduced by key to get its absolute count. In addition, to inspect the resulting counts we can retrieve all the elements of the RDD, by recalling the `collect()` action, that would be provided as a list of n number of tuples each containing two elements (i, c), namely the hashed item i and the corresponding count c. The three most frequent items in the baskets are hashed with the following integers: $4924, 2024$ and 4354, which are associated to the actors name "Brahmanandam", "Adoor Bhasi" and "Matsunosuke Onoe" and with the absolute counts $798, 585, 565$ corresponding to the number of movies they played in, respectively.

Now that we have the absolute counts for each singleton (each actor alone), we need to construct a frequent-items table, with the format of an array, that would help us identifying the frequent items according to a specified threshold s. This sequence of integers would follow the orders of the items in the hash table and would assign to each of them -1 if the actor in that position is not frequent or the hashed integer if he/she is actually frequent. To be clear, here we decided to use -1 instead of zero to track non-candidate items, in contrast with what was described in the A-Priori phases, to be consistent with the default indexing of Python/PySpark where the first element corresponds to the position 0. With respect to the threshold s, the lower it gets the greater becomes the set of frequent items, which requires our algorithm to read over larger amount of elements in each basket during the next passes.

Coherently, since we are mostly interested in the "relevant" frequent itemsets rather than all of them, we decided to tailor the value of support threshold s on the basis of the *boxplot* of the distribution of counts. On average, the number of movies for which the actors/actresses played in is equal to $\mu \approx 3$, thus we expect to see the majority of the values concentrated on the lower tail corresponding to the minimum of the distribution, while a relative "small" number closer to the maximum. In fact, the boxplot and the histograms displayed in Figure 1 and Figure 2 confirm that the distribution of movies played by each actor is not symmetric, but positively skewed, and it could be seen that there is the presence of outliers detected for support values larger than 5. Therefore, considering that the number of movies in file is massive and we are interested in rare frequent itemsets, we decide to set the support threshold at $s = 10$ corresponding to the 95th percentile of the distribution, which could still include a large number of candidate elements but at least we are sure that the goal of the research would be achieved.



(a) Boxplot of the absolute counts for the itemsets of size $k = 1$ (singleton)



(b) Histogram of the absolute counts for the itemsets of size $k = 1$ (singleton)

In practice, the threshold is set high enough in order to have a meaningful frequent itemsets analysis, since we need the final result, in terms of frequent itemsets, to be smaller in magnitude relatively to all the distinct elements and the possible combinations we could get. Coming back to the threshold value, this is equivalent to say that every combination of actors, it could be singleton or pairs, triples, etc., to be considered as a candidate itemsets needs to have played in more than 10 movies overall. At this point, we can substitute the non-candidate items in the hashed baskets file with -1 and we remove them, as well as the baskets that turned out to be empty or composed by a single item since they only contained non-frequent elements. Now, the second pass of the A-Priori algorithm can start. The aim of this stage is to read again all the items in the baskets and obtain the counts of every combination that consists in two frequent items (or doubleton). Therefore, we would parallelize the list of baskets and for each of them we would compute and map the combination of frequent pairs, using the `combinations()` function of the *itertools* library, and we would the reduced by key to get the overall counts.

The obtained absolute frequencies are returned as a huge list of tuples, each of them containing two elements: one tuple (i, j) corresponding to the pair items i, j and the corresponding count c overall the baskets. In alternative, one can try to adopt the *Triples method* that consists in storing each pair (i, j) and the corresponding count c as a single triple (i, j, c), which is more memory efficient since we just store three integers for every pair, in general for set of elements of size k we store $k + 1$ integers in the representing tuples datatype. This time, to inspect the three most frequent pairs of actors overall the baskets, we would parallelize the list of counts and map the items from their hashed value to their corresponding actor name. In particular, those three pairs are: "Adoor Bhasi - Prem Nazir", "Bahadur - Adoor Bhasi" and "Jayabharati - Adoor Bhasi" and their equivalent counts of 237, 169 and 162 number of movies they played in, respectively.

As we did in the previous pass, we construct another hash table in the form of an array, where each element corresponds to a couple of items, and we substitute -1 to the non-frequent pairs if their corresponding count c is less than the threshold $s = 10$. Now, after we remove the negative integers from the array of pairs, we get all the frequent itemset of size $k = 2$ that would be unwinded and the duplicate items removed. For this reason, among all the frequent pairs we end up with **+++** distinct items that would be the only ones to populate the baskets for the third pass of the A-Priori algorithm, when we would aim to find frequent itemsets of size $k = 3$ (triples). Hence, we would repeat the steps as before by removing non-frequent items from the baskets file and the baskets left with less than three elements, so that we obtain the input file with all frequent baskets. At this time, since the procedure represented by the next chunks of code is equivalent to the one performed up to this point, we would rather build a function able to return the counts of the itemsets of any size k*, even larger than $k \in [1, 4]$. Therefore, we would bundle the map and reduce operations corresponding to the passes of the A-Priori algorithm, in a single function:`APriori()`, which would take the following arguments:

- **hashed_baskets**: list; baskets file under the form of a list where the elements of each itemset are formatted to their corresponding hash values for example: $[[0, 1, 2], [3, 4, 5], ..., [..., n]]$.
- **k_size**: int; the size k of the itemsets for which the counts should be computed, for example `k_size = 2` corresponds to pairs, `k_size = 3` to triples, etc.
- **s_threshold**: int; the value of the support threshold to be used for determining frequent and non-frequent itemsets for `k_size > 1`.

Consequently, we would use this function to find frequent triples and quadruplets and we would then check whether the results are consistent with the manual procedure of the several map and reduce phases, which of course would more time to be executed. In this setting, we would give as argument of the function the whole baskets file, which corresponds to the original one, and the parameter $k \in [3, 4]$ corresponding to the size of the itemsets and the same support threshold used in the previous analysis. Hence, for the manual procedure we would pass another time over the data by mapping and reducing by key the combinations of triples to count their absolute frequencies, which would be compared with the same threshold value $s = 10$. Accordingly, once we format the hashed integers with their corresponding actors name, we find that the results of the function and the ones of the manual pass are identical and and above all we got that the three most frequent triples are: "Kitsuraku Arashi - Kijaku Ôtani - Matsunosuke Onoe", "Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe" and "Kitsuraku Arashi - Suminojo Ichikawa - Matsunosuke Onoe" that acted in 122, 100 and 95 movies together, respectively.

In the very last part of the analysis, since there is some free space in the main memory of the free virtual machine provided by GoogleColab, we decide to search for frequent itemsets of size $k = 4$. As before, we first recall the `APriori()` function and then we repeat the same procedure as before, by removing the non-frequent items from the remaining baskets and filtering off the ones with length less than 4. Next, once we map the combinations of four elements in each basket and we reduce over the counts, we get the support of all the itemsets of cardinality $k = 4$, where the three most frequent quadrupltes overall the baskets are: "Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe", "Kitsuraku Arashi - Sen'nosuke Nakamura - Kijaku Ôtani - Matsunosuke Onoe" and "Sen'nosuke Nakamura - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe" that appeared in 86, 62 and 54 movies together, respectively. Of course, when we inspect frequent itemsets of larger size we know that not only the overall number of baskets left with frequent elements decreases, but also the resulting counts are likely to be lower than the absolute frequency of the itemsets of size $k - 1$, or at maximum that specific value. Therefore, if one is interested in frequent itemsets of larger cardinality, then the function built in the code available on the Github folder could be useful, since it only needs to execute the first chunks of code up to the construction of the hashed baskets file, with the `k_size` argument that would take care about the cardinality of the itemsets of interest.

5 Comments on the Experimental Results

In this section, we would inspect the frequent itemsets obtained at each pass of the A-Priori algorithm and we would give reasoning about the results, where the top 10 most frequent itemsets for each size of $k \in [1, 4]$ are displayed in the tables contained in the *Appendix*. In this setting, Table 2 shows the ten most frequent single actors overall the IMDb dataset, where the first two positions are occupied by two Indian artists, "Brahmanandam" and "Adoor Bhasi", first and second with 798 and 585 respectively, while the third place is owned by the Japanese performer "Matsunosuke Onoe" (565 movies). Given that the dataset was distributed under the IMDb non-commercial licensing/copyright, we could try to search for some additional information about those actors by querying the IMDB website and see whether our results are actually consistent or not. Surprisingly, we find that "Brahmanandam" has 1.175 credits as actor, he is still in activity and holds the Guinness World Record for having acted in the highest number of films (also in a single language). Moreover, the second most frequent actor "Adoor Bhasi" is credited in 624 movies, even though Wikipedia states that he played in more than seven hundred movies, but we are glad to see that he is frequently mentioned alongside with "Prem Nazir" (the fifth most frequent actor in our ranking). Lastly, "Matsunosuke Onoe" is cited in 948 performances in the IMDb dataset (more than one thousand for Wikipedia), even though the majority of his works were short films, this is likely to be the reason why his support threshold is the third overall.

Accordingly, even though we found a lower number of movies played by "Brahmanandam" and the other two actors on the podium with respect to the credits of the IMDb website, we are pretty confident that the baskets file that we had at disposal is in fact a subset of the records stored in the whole database, which of course is not available for free and has its own commercial license/copyright. Therefore, we could impute the difference between the support thresholds, found by our implemented system finding frequent itemsets, and the credits of the filmography, edited in the IMDb website, due to the sampling. Continuing the review of the results, the most frequent pair of actors we found is composed by "Adoor Bhasi" and "Prem Nazir" (237 movies), which are often mentioned together as Wikipedia specified, where the former is present in three other itemsets among the first ten, while we do not find the record-breaking actor "Brahmanandam". At first sight, we could justify this result by simply highlighting "Adoor Bhasi" and "Prem Nazir" shared the screen during the same acting years, while "Brahmanandam" is thirty years younger and it could be that he played in fact with a larger audience in earlier ages (from 1987 - on), so that his support value is distributed among least frequent artists. In addition, we find that the average number of actors that played with "Brahmanandam" in each basket is $4.25 \approx 4$, which almost coincide to the mean value for all the baskets $\mu = 4.3 \approx 4$. Hence, we could think to impute the absence of the first ranked actor in the frequent pairs mostly due to the small number of peers overall the baskets, which decreases the probability of working with any of them several times. In addition, the third ranked actor "Matsunosuke Onoe" appears in the fourth most frequent couple with "Kijaku Ôtani" with 147 appearances together and two other frequent itemsets in the top ten ranking, with 126 and 113 movies played alongside, respectively.

Now, by inspecting Table 4 and Table 5 we see that the support values of the triplets and quadruplets, namely the frequent itemsets of size $k = 3$ and $k = 4$ respectively, are fairly lower than the ones of the two previous tables, as the monotonicity property clearly explains. Moreover, among the most frequent triples we clearly see that eight out of ten are solely composed by Japanese actors, where "Matsunosuke Onoe" appears in the first place with "Kitsuraku Arashi - Kijaku Ôtani" in 122 movies. On the contrary, the only two Indian triples are composed by "Jayabharati - Bahadur - Adoor Bhasi" and "Adoor Bhasi - Prem Nazir - Thikkurisi Sukumaran Nair" that acted in 75 and 74 movies together, with "Thikkurissy Sukumaran Nair" who ended up in being the master of "Prem Nazir" as Wikipedia states. In the final part, the last table contains only Japanese actors as records for top ten raking of most frequent quartet, where "Matsunosuke Onoe" appears in seven frequent itemsets of size $k = 4$, with the combination "Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe" that occupies the first place with 86 movies played by these four performers.

6 References

Written sources
Leskovec, Jure , Rajaraman, Anand & Ullman, Jeff. (2014). *Mining of Massive Datasets*. Palo Alto, California.

Online Website sources
Ratings, Reviews, and Where to Watch the Best Movies & Tv Shows. Internet Movie Database (IMDb), 13 September 2021, `www.imdb.com`.

Wikipedia, the free encyclopedia, Wikimedia Foundation, 13 September 2021, `https://en.wikipedia.org/wiki/Main_Page`.

7 Appendix

Frequent Itemsets	Support Value
Brahmanandam	798
Adoor Bhasi	585
Matsunosuke Onoe	565
Eddie Garcia	507
Prem Nazir	438
Sung-il Shin	411
Paquito Diaz	391
Masayoshi Nogami	387
Mammootty	381
Aachi Manorama	355

Table 2: Top 10 frequent itemsets with cardinality $k = 1$ and support threshold $s = 10$

Frequent Itemsets	Support Value
Adoor Bhasi - Prem Nazir	237
Bahadur - Adoor Bhasi	169
Jayabharati - Adoor Bhasi	162
Kijaku Ôtani - Matsunosuke Onoe	147
Kitsuraku Arashi - Matsunosuke Onoe	126
Adoor Bhasi - Thikkurisi Sukumaran Nair	122
Kitsuraku Arashi - Kijaku Ôtani	113
Suminojo Ichikawa - Matsunosuke Onoe	113
Jayabharati - Bahadur	109
Panchito - Dolphy	103

Table 3: Top 10 frequent itemsets with cardinality $k = 2$ and support threshold $s = 10$

Frequent Itemsets	Support Value
Kitsuraku Arashi - Kijaku Ôtani - Matsunosuke Onoe	122
Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	100
Kitsuraku Arashi - Suminojo Ichikawa - Matsunosuke Onoe	95
Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa	87
Sen'nosuke Nakamura - Kijaku Ôtani - Matsunosuke Onoe	80
Jayabharati - Bahadur - Adoor Bhasi	75
Adoor Bhasi - Prem Nazir - Thikkurisi Sukumaran Nair	74
Kitsuraku Arashi - Sen'nosuke Nakamura - Matsunosuke Onoe	70
Hôshô Bandô - Ritoku Arashi - Enshô Jitsukawa	69
Sen'nosuke Nakamura - Suminojo Ichikawa - Matsunosuke Onoe	64

Table 4: Top 10 frequent itemsets with cardinality $k = 3$ and support threshold $s = 10$

Frequent Itemsets	Support Value
Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	86
Kitsuraku Arashi - Sen'nosuke Nakamura - Kijaku Ôtani - Matsunosuke Onoe	62
en'nosuke Nakamura - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	54
Kitsuraku Arashi - Sen'nosuke Nakamura - Suminojo Ichikawa - Matsunosuke Onoe	51
Hôshô Bandô - Ritoku Arashi - Shôzô Arashi - Enshô Jitsukawa	51
Chosei Kataoka - Kitsuraku Arashi - Kijaku Ôtani - Matsunosuke Onoe	48
Kitsuraku Arashi - Utae Nakamura - Kijaku Ôtani - Matsunosuke Onoe	46
Chosei Kataoka - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	45
Kitsuraku Arashi - Sen'nosuke Nakamura - Kijaku Ôtani - Suminojo Ichikawa	45
Kitsuraku Arashi - Sen'nosuke Nakamura - Kijaku Ôtani - Suminojo Ichikawa	44

Table 5: Top 10 frequent itemsets with cardinality $k = 4$ and support threshold $s = 10$