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 able to find frequent itemsets of different cardinality. The main source of data that we would consider 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 and reconstruct the useful associations. The task of the project consists in writing from scratch the phases of one algorithm studied throughout the course in *Algorithms for Massive Datasets* of the academic year $2020/2021$, considering "actors" as items and "movies" as baskets. In this setting, the framework used to develop and test the code is provided by *Google Colab* and consists in a free hosted runtime that exploits Google Drive personal accounts to establish the connection with the Kaggle API to download and store the IMDb dataset, provided that my personal tokens are uploaded on the notebook. Of course, the main code that carries out the task of the project should be developed 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.

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 table files, which attributes would be shortly described. Moreover, we would mention the key fields of the three "*.tsv" files, which would be used later on to build-up the baskets list to be fed as input of our frequent itemsets algorithm, and perform some exploratory data analysis and data pre-processing tasks. 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 the its main characteristics and differences with respect to an alternative family 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 the support threshold to be able to achieve meaningful and interesting results according to our expectations. 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 referred of the iterative procedure, and would give some rational arguments to justify the criterion used to discriminate between frequent and non-frequent itemsets. Coherently, in the last chapter we would inspect the ranking of the discovered frequent itemsets of different size, from singleton to quadruplets, and we would try to provide some evidences that validate our findings by relying on the additional information on the *Wikipedia* and *IMDb* websites.

The main reference used for the realization of this study is the book *Mining of Massive Datasets*, which was suggested since the beginning of the course, that is written by A. Rajaraman e J. Ullman and is freely available on the website of the authors[1]. In addition, 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[2]. Accordingly, for a deeper understanding of the code and the tasks implemented throughout the analysis it is suggested to open the Python notebook directly on Github and follow the main code alongside with this document, where every stage and obtained results are interpreted and contextualized.

[1] *Mining of Massive Datasets (Rajaraman & J. Ullman 2014)*

[2] Github folder of the project

2 Description of the Dataset

The dataset used throughout the analysis is taken from Kaggle[3] 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 my API credentials (username and associated key) required to import the dataset remotely. 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. Once the JSON file is uploaded, the compressed file is unzipped and removed, with the three "*.tsv" (tab-separated values) files that are assigned to three variables corresponding to three DataFrames: *titles*, *job* and *actors*.

2.1 Exploratory Data Analysis

The three versions of the datasets used in this project are filtered with respect to a subset of the original features, thus we selected only the ones suitable for our task which are provided in the 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 titleId.
- **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.

[3] IMDb Dataset

Along with the import of the dataset and the filter of the selected variables, the *Nan* values are removed, so that the three working datasets 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 just movies.

Coherently, we retrieved the two datasets, one with all the actors and the other with movies only, to perform a join based on the movies each actor played in, then we grouped the observations by the primary title of the movies and we retrieved the number of actors for each screenplay. In particular, the three movies with the most number of actors are *Hamlet*, *Macbeth* and *Honeymoon* with 145, 140 and 139 characters, respectively. Then, apart from these three ranked movies, only other sixteen ones have at least one-hundred actors, while considering all the titles there are on average 5 actors per-movie.

2.2 Data Pre-processing

At this stage, we would proceed by manipulating a bit the data points so that they are ready to be given as input of our frequent itemsets system. Therefore, we would initially group the observations by primary title and we would collect in a list all the actors belong 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). In addition, we could represent the set of baskets as a dictionary having as keys the titles of the movies and as values the corresponding list of actors, but still we would in any case deal with a set of baskets with cardinality equal to 346.874 movies.

However, given that strings have more memory footprint rather than integers, we would like to represent 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 to each of them starting from zero. Accordingly, we would build sort of a hashing table by creating a dictionary of size 560.653, 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 is 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 that could have different length 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, 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 could be sure that is not possible to have a frequent quartets or larger sets. This fact is related to a factual conclusione defined as *Monotonicity Property* of the itemsets, which basically states implicit coming from the observations 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 largest frequent subset given that 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 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 they 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 or 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 is able to differentiate them lies in the number of times they pass through the basket file, which 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 the elements to search for frequent itemsets of different size k 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*, 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 and were eliminated. Consequently, the second pass works by mapping each basket and looking for all pairs of frequent items, then by reducing them adding one 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 and all this procedure can be repeated for further analysis if frequent itemset of larger size k are useful to inspect.

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 $k \in [1, 3]$ 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 be interested in. 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 would 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 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 access 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 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 a hash table, which stored 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 are provided as a list n number of tuples each containing two elements (i, c), namely the hashed item and the corresponding count. The three most frequent items in the baskets are hashed with the following integers: $799, 5210, 7397$, which are associated to the actors name "Brahmanandamm", "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 items 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, which is displayed in Figure 1. On average, the number of movies for which the actors/actresses played in is equal to $\mu \approx 3$, thus most of the values are concentrated on the lower tail corresponding to the minimum of the distribution, while there a relative "small" number of observations above the 75th percentile. Therefore, considering that the number of movies in file is massive and the square of the boxplot (values between the first and third quartiles) is barely visible, we decide to set the threshold exactly at $s = 10$ corresponding to the 95th percentile, which could still include a large number of candidate itemsets but we are sure that the goal of the research would be achieved.

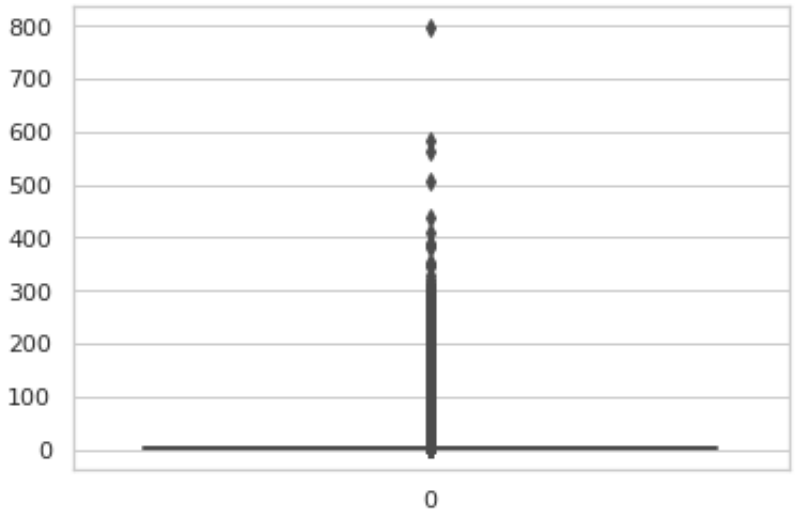

Figure 1: Boxplot of the absolute counts for the itemsets of size $k = 1$ (singleton)

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 singleton or pairs, triples, etc., to be considered as a candidate itemsets needs to be included in more than 10 baskets 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, so that 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, that would be reduced by key to get the overall counts.

The resulting 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 "Kijaku Ôtani - Matsunosuke Onoe" and their equivalent counts of 237, 169 and 165 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 2397 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 thus the baskets that consists of less than three elements, so that we obtain the input file with all frequent baskets. As before, we 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, the three most frequent triples found overall the baskets are: "Kitsuraku Arashi - Kijaku Ôtani - Matsunosuke Onoe", "Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe" and "Kitsuraku Arashi - Suminojo Ichikawa - Matsunosuke Onoe" that acted in 121, 106 and 101 movies together, respectively.

In the very last part of the analysis, since there is some free space in the main memory of the virtual machine provided by GoogleColab, we decide to search for frequent itemsets of size $k = 4$. As before, we repeat the same procedure, by removing the non-frequent items from the remaining baskets and filtering off the ones with length less than 4, and we map the combinations of four elements in each basket. Then, by reducing over the tuple of items we get the absolute counts of each element set of cardinality $k = 4$, where the three most frequent itemsets are 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 91, 66 and 57 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 $k - 1$, in this case triples. Therefore, if one is interested in frequent itemsets of larger cardinality, then the code available in the Github folder could be executed for free on GoogleColab, by copy-pasting the chunks of code corresponding to the two main pass of the A-Priori algorithm and change the size of the combinations to be created.

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 give reasoning about the results, where the top 10 most frequent itemsets of size $k \in [1, 4]$ are displayed in the tables of the next chapter (*Appendix*). In this setting, Table 1 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[4] 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/copyrights. 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 is a couple 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 $6.12 \approx 6$, which is slightly above the mean value for all the baskets $\mu = 4.88 \approx 5$. 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" that appears in the fourth most frequent couple with "Suminojo Ichikawa" with 136 appearances together and two other frequent itemsets in the top ten ranking, with 120 and 106 movies played alonside, respectively.

Now, by inspecting Table 3 and Table 4 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 surely justify. Moreover, among the most frequent triples we clearly see that nine out of ten are solely composed by Japanese actors, where "Matsunosuke Onoe" appears in the first place with "Kitsuraku Arashi - Kijaku Ôtani" in 121 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 eight frequent itemsets of size $k = 4$, with the combination "Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe" that occupies the first place with 91 movies played by these four performers.

[4] IMDb website

6 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 1: 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
Kitsuraku Arashi - Matsunosuke Onoe	136
Adoor Bhasi - Thikkurisi Sukumaran Nair	123
Suminojo Ichikawa - Matsunosuke Onoe	120
Kitsuraku Arashi - Kijaku Ôtani	113
Jayabharati - Bahadur	109
Panchito - Dolphy	106
Sen'nosuke Nakamura - Matsunosuke Onoe	106

Table 2: Frequent itemsets with cardinality $k = 2$ and support threshold $s = 10$

Frequent Itemsets	Support Value
Kitsuraku Arashi - Kijaku Ôtani - Matsunosuke Onoe	121
Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	106
Kitsuraku Arashi - Suminojo Ichikawa - Matsunosuke Onoe	101
Sen'nosuke Nakamura - Kijaku Ôtani - Matsunosuke Onoe	88
Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa	87
Jayabharati - Bahadur - Adoor Bhasi	75
Kitsuraku Arashi - Sen'nosuke Nakamura - Matsunosuke Onoe	75
Adoor Bhasi - Prem Nazir - Thikkurisi Sukumaran Nair	74
Hôshô Bandô - Ritoku Arashi - Enshô Jitsukawa	70
Sen'nosuke Nakamura - Suminojo Ichikawa - Matsunosuke Onoe	68

Table 3: Frequent itemsets with cardinality $k = 3$ and support threshold $s = 10$

Frequent Itemsets	Support Value
Kitsuraku Arashi - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	91
Kitsuraku Arashi - Sen'nosuke Nakamura - Kijaku Ôtani - Matsunosuke Onoe	66
en'nosuke Nakamura - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	57
Kitsuraku Arashi - Sen'nosuke Nakamura - Suminojo Ichikawa - Matsunosuke Onoe	54
Chosei Kataoka - Kitsuraku Arashi - Kijaku Ôtani - Matsunosuke Onoe	52
Hôshô Bandô - Ritoku Arashi - Shôzô Arashi - Enshô Jitsukawa	52
Kitsuraku Arashi - Utae Nakamura - Kijaku Ôtani - Matsunosuke Onoe	50
Chosei Kataoka - Kijaku Ôtani - Suminojo Ichikawa - Matsunosuke Onoe	49
Chosei Kataoka - Kitsuraku Arashi - Suminojo Ichikawa - Matsunosuke Onoe	48
Kitsuraku Arashi - Sen'nosuke Nakamura - Kijaku Ôtani - Suminojo Ichikawa	45

Table 4: Frequent itemsets with cardinality $k = 4$ and support threshold $s = 10$