Movie-actor market-basket analysis
Project of Algorithms for Massive Datasets

Mattia Lecchi (964860)

February 13, 2025

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 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 or any other person for assessment on this or any other course of study.

1 The dataset

The dataset in use for the analysis is a dump of Letterboxd from June 26, 2024. The only needed table in the dataset for the market-basket analysis on movie and actors is the `actors.csv` file, which links those two entities: the ”id” column contains the movie identifier, the ”name” one the actor full name.

1.1 Preprocessing

The neeeded representation of the provided data is a list of baskets, that is a list of tuples of actors who took part in the same movie. To obtain this representation, the rows must be grouped by movie id, keeping only the actor names. This operation can be easily parallelized using Apache Spark and the MapReduce model.

2 Algorithms implementation

The proposed algorithm for the analysis is the Savasere, Omiecinski and Navathe (SON), implementing the Park, Cheng and Yu (PCY) algorithm for retrieving frequent itemsets from the chunks. As the actual memory intensive task is finding candidate pairs, the apriori algorithm is suitable for the generation of candidate itemsets of cardinality greater than 2.

2.1 PCY

For the implementation of the PCY algorithm, some data structures can be defined.

Baskets iterator Baskets files are constructed via Spark and saved on disk in a number of parts, as described in section 1. For this implementation, the PCY algorithm works on a single machine. It must open one chunk file at a time and load baskets in a lazy way. This aspect should be transparent to the algorithm implementation.

For this purpose, an iterator class is defined; the class manage the opening of the chunk files and the retrieval of each basket implementing the Python iterator protocol.

Bitmap A data structure needed for the PCY algorithm is the bitmap, useful for the compressed representation of candidate pairs. A list of boolean values is converted in a sequence of bytes through the `packbits` method provided by the `numpy` module.

2.1.1 Algorithm implementation

In the first basket scan, the algorithm:

- counts all item occurrences, storing items and counts in a hash table;
- counts all hash values of couples in each basket, that is, fixed an hash function on couples $h : X \times X \to \{0, ..., m-1\}$ and a 32-bits integers vector V of size m initialized with zeros, when a couple $(x, y) \in X \times X$ is encountered, the algorithm increment by 1 the value $V_{h(x,y)}$.

The hash function implemented takes an itemset of arbitrary length, compute the hash of each item using the Python `hash()` built-in method and XOR the results, so that the actual ordering of the tuple of items doesn't affect the final hash value. The result is finally returned modulo m.

After the first pass, the algorithm create a bitmap B of length m with 1s in the same position as frequent buckets, namely the ones exceeding a fixed threshold s. The items list is filtered, discarding all non frequent items. Both V and the item counters are deleted from main memory. The resulting bitmap is 1/32 of size with respect to the vector of bucket counters, as every 32-bits integer is replaced by a single bit. This bitmap is used to create each couple of frequent items and filter out those that don't hash in a frequent bucket, that is the algorithm generates pairs of frequent items (x, y) and returns the ones such that $B_{h(x,y)} = 1$.

Actually, not all returned values are frequent couples, there can be false positives due to the fact that an infrequent pair can hash to the same bucket of a frequent pair. Another baskets scan is needed to drop false positives.

The algorithm finds only frequent pairs; the search can be continued using the apriori algorithm, iteratively passing through basket files for each size, until a size s is reached such that frequent itemsets don't exist. Candidates of size s for each basket are obtained by the combinations of the frequent items in a basket that appears at least $s-1$ times in frequent itemsets of size $s-1$. For example, the itemset $\{a, b, c\}$ with cardinality equal to 3 is a candidate if and only if $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$ are frequent, that is, each of a, b and c appears at least 2 times in the frequent itemsets of size 2.

2.2 SON

While the PCY algorithm alone can be used to find frequent itemsets, the objective is to use it with SON, obtaining an algorithm suitable for distributed environments, without yielding false positives. Each computing node will run PCY on a chunk of the baskets.

2.2.1 Memory usage

An important aspect is the choice of the right size of the vector of bucket counters to obtain a low number of false positives while avoiding thrashing. The main elements to store in memory are:

- the hash table of item counters;
- the array of bucket counters;
- the bitmap of frequent buckets.

The hash table is the only large structure to consider for correctly sizing the number of buckets, since the bitmap will replace the counters later. However, it's not easy to predict a priori the size of an hash table, therefore it will be measured with the help of the `pympler` module, after all items across all baskets have been counted. It must be noted that the resulting size strongly depends on many factors, like Python version, operating systems and CPU architecture, so the value should be interpreted as a rough approximation. Considering a Google Colab CPU environment, the observed size is 172.233 MB, so it's assumed that the hash table won't occupy more than 200 MB of main memory, leaving 1.8 GB free. The maximum number of buckets with 32-bits integer that fits in main memory is:

$$\frac{2 \cdot 10^9 \text{ B} - 2 \cdot 10^8 \text{ B}}{4 \text{ B}} = 4.5 \cdot 10^8 \tag{1}$$

After the compression, the resulting bitmap size is $\frac{1.8\text{GB}}{32} = 56\text{MB}$.

2.2.2 Algorithm implementation

The implementation is composed of two subsequent tasks, both parallelizable on different machines via Apache Spark:

- The first task applies the PCY algorithm on a number of chunks c via the `mapPartitions()` method on the baskets RDD, which apply a map function on a chunk instead of a single key-value pair. The threshold used is s/c. Each node returns a list of candidates which could contain false positives. Duplicates are removed via the `distinct()` method and finally the whole list of candidates is collected by a node.

- The second task again applies a map function on chunks, this time counting all occurrences of candidates in the assigned chunk. The result of this mapping is a list of itemset-count couples, which are then grouped by itemset. Finally, the counts are summed; if the sum is below the threshold, then the itemset is a false positive, so it's discarded. The resulting itemsets are returned with their counts.

The list of candidates must fit in main memory. If it's not the case, either the number of false positives is too large, thus a larger number of buckets is needed, or the actual frequent itemsets are too many, hence an higher threshold must be specified.

False positives exist not only because PCY provides them, but also because an itemset frequent in a chunk can be infrequent in the whole dataset.

3 Results of the analysis

The SON algorithm is executed on the whole dataset, considering a frequency threshold of 100, 5 chunks of baskets and a number of buckets equal to 10^8. The results of the analysis are 78 itemsets, the most frequent one is the couple Larry Fine and Moe Howard with 236 appearances. Among the frequent itemsets, there are 54 duets, 17 and 6 are the triplets and the quartets respectively, and only 1 quintet. The largest sets are mainly composed by members of famous bands like Pearl Jam, Beatles and Metallica.

PCY can be used as a standalone algorithm on the whole dataset, recalling that another pass on the basket files is needed to drop false positives. Considering 100 as frequency threshold and 10^8 buckets, the algorithm finds 86 frequent items, thus yielding 8 false positives. After making another pass counting occurrences of the found candidates, the results are the same 78 frequent itemsets as SON.

4 Conclusions

The implemented algorithms provides good scalability in Big Data contexts. It's assumed that the number of distinct singletons is small enough to let the item counters fit in main memory. The tricky part is the counting of couples; in fact, while the number of singletons is usually relatively small, this is not the case for couples, as the number of integers needed is almost the square of the number of singletons. Triples generated from frequent couples are usually low in number, as the vast majority of couples is infrequent, so memory consumption issues with larger itemsets don't arise.

The apriori algorithm reduce the number of couple counters to the square of the number of frequent singletons, while PCY address the problem of too many counters by hashing couples in a fixed number of buckets, thus using an integer vector of fixed size. However, if the number of buckets is too low, the algorithm will generate a huge number of candidates, potentially filling up the main memory. If more machines can be used, the SON algorithm can help distributing the load on different computing nodes. The algorithm can be applied in conjunction with PCY on basket chunks to obtain at a computing node level the advantages described above. However, one thing to consider in this setting is the number of false positives generated for each chunk. Even if the used algorithm were exact (like apriori), a frequent itemset in a chunk could be infrequent in the dataset and dropped in the next stage of SON. The number of false positives is amplified by PCY, as it yields false positives in itself, thus a low memory usage on PCY execution is paid in the subsequent stage of false positives filtering.