Movie-actor market-basket analysis

Project of Algorithms for Massive Datasets

Mattia Lecchi (964860)

January 30, 2025

1 The dataset

The dataset in use for the analysis is a dump of Letterboxd from June 26, 2024. The only useful 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.

Preprocessing For a market-basket analysis, a useful representation of the provided data is a list of baskets, that is 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 by 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 in 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. 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 constructed; 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 values in hash tables that associate the items to the frequencies;
- 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 element 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 hash value.

It's then possible to create a bitmap B with ones in position of frequent buckets and retain only frequent itemsets, namely the ones exceeding a fixed threshold s. The items list is filtered, discarding all non frequent items and both the couples 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 we replaced every 32-bits integer with a single bit. This bitmap can be used to create each couple of frequent items and filter out those that don't hash in a frequent bucket, that is returning pairs (x, y) 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 basket scan would be needed to drop false positives.

The algorithm find only frequent couples; the search can continue using the apriori algorithm and making a pass of basket files for each size incrementally, until a size s such that frequent itemsets don't exist is reached.

2.2 SON

While the PCY algorithm alone can be used to find frequent itemsets, the objective is to use it with SON, obtaining a suitable algorithm for distributed environments, without 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 size of the hash table is the only big structure to consider to correctly size the number of buckets, since the bitmap will bereplace the counters in a second moment. 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 from 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 will 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 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 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, each of them 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 Spark RDD, which apply a map function on a chunk instead of a single key-value pair. The threshold used is cs. Each node returns a list of candidates which potentially contains false positives. Duplicates are removed via the `distinct()` method and finally the whole list of candidate is loaded.

- The second task again applies a map function on chunks, this time counting all occurrences in the assigned chunk considering the whole list of candidates returned by the previous task. The result of this mapping is a list of couples itemset-count. The results are grouped by itemset and the counts are summed; if a total count is under the threshold, the associated itemset is discarded, as it was a false positive. Lastly, the resulting itemsets are returned with their counts.

It's assumed that the list of candidates fits in main memory. If it's not the case, either the number of false positives is too large, thus a larger number of buckets in PCY is needed, or the itemsets that exceed the fixed threashold are too many, hence an higher threshold must be specified.

False positives are present not only because PCY provides them, but also because an itemset frequent in a chunk can be infrequent in the whole dataset. That is, there is a chunk in which the count is greater than cs but the total number of occurrences is lower than s.

3 Results