Project 2: Market-Basket Analysis
"Old Newspapers" dataset

December 15, 2022

Authors: Mathias Cardarello Fierro, Lorenzo Polli

Students number: 963346, 966293

Course: Algorithms for Massive Data

Università degli Studi di Milano

Contents

[1]This section is based on Chapter 6 of the textbook “Mining of Massive Datasets” written by A. Rajaraman and J. Ullma.

Introduction

This project aims at investigating techniques generally used to conduct market-basket analysis over huge datasets in order to find frequent itemsets.

In this specific case, the dataset is taken from the public repository Kaggle and contains more than 16 million of paragraphs about old newspapers. Although the newspapers were written in 67 different languages, for the scope of the research, the analysis is conducted over English newspapers only. In total, a subset of more than 1 million articles published between the years 2005 and 2012 were analyzed, using algorithms for massive datasets. The dataset is composed by the following attributes:

- Language: the language in which the sentence is expressed.
- Source: from which newspaper the text was taken from.
- Date: information about the article's release date.
- Text: the sentence or the paragraph sampled from the newspaper's article.

```
+--------+------------+----------+--------------------+
|Language|      Source|      Date|                Text|
+--------+------------+----------+--------------------+
| English| latimes.com|2012/04/29|He wasn't home al...|
| English|stltoday.com|2011/07/10|The St. Louis pla...|
| English|   freep.com|2012/05/07|WSU's plans quick...|
| English|      nj.com|2011/02/05|The Alaimo Group ...|
| English|  sacbee.com|2011/10/02|And when it's oft...|
+--------+------------+----------+--------------------+
only showing top 5 rows
```

The work is developed in the following way: the methodology applied to analyze the dataset is described in the next sections as well as the algorithms implemented. To achieve the results, Python and Apache Spark were used. Apache Spark is a distributed computing framework for large-scale data processing whose potentialities in terms of scalability are studied. Finally, the results are compared and discussed.

1 Theoretical framework[2]

1.1 A-Priori Algorithm

The market-basket model of data is used to describe a common form of many relationship between two kinds of objects, items and baskets or "transactions" that consist of a set of items (itemset). The problem of finding the most frequent itemsets is equal to get sets of items that appear in many of the same baskets. The model can be applied to many kinds of data, such as text articles, in which items are words and the baskets are the sets that contain those words that are present in the document.

A basic approach for finding frequent itemsets is the A-Priori Algorithm, which works by eliminating most large sets as candidates by looking first at smaller sets and recognizing that a large set cannot be frequent unless all its subsets are, that is, the *monotonicity* property:

- If a set I of items is frequent, then so is every subset of I.

If we have enough main memory to count all sets, we could find the most frequent itemsets with an exhaustive count, in a single pass. The A-Priori Algorithm is designed to reduce the number of sets that must be counted, at the expense of performing two passes over data.

The first pass consists in creating two columns: one translates the item as an integer and the other one as an array of counts. The ith array element counts the occurrences of the item at index i. The former column is useful to have a word-to-number correspondence to be able to encode each word with the respective integer count. The latter column counts the frequency of singletons, thus each element contains the number of occurrences of the respective item.

After the first pass, we examine the counts of the items to determine which of them are frequent as singletons. In order not to obtain too many frequent sets, we should set a high threshold s such as 1% of the baskets. For the second pass of A-Priori, we create a frequent-items table that consist in an array, and the entry for i is either 0, if item i is not frequent, or a unique integer in the range 1 to m if item i is frequent.

The second pass can be described as follows.

1. For each basket, look in the frequent-items table to see which of its items are frequent.

2. In a double loop, generate all pairs of frequent items in that basket.

[2]This section is based on Chapter 6 of the textbook "Mining of Massive Datasets" written by A. Rajaraman and J. Ullma.

3. For each such pair, add one to its count in the data structure used to store counts.

Finally, at the end of the second pass, examine the structure of counts to determine which pairs are frequent.

The construction of the collections of larger frequent itemsets and candidates proceeds until at some pass we find no new frequent itemsets and stop. That is:

1. Define C_k, to be all those itemsets of size k, every $k-1$ of which is an itemset in L_{k-1}, the set of truly frequent itemsets of size $k-1$.

2. Find L_k by making a pass through the baskets and counting all and only the itemsets of size k that are in C_k. Those itemsets that have count at least s are in L_k.

In summary, the A-Priori Algorithm alternates between constructing candidate sets and filtering to find those that are most frequent, with the help of the monotonicity property. Even though the algorithm may seem simple to implement, it is not free of drawbacks. In particular, at each step, candidate sets must be built, and to do it the algorithm repeatedly scans the database. Furthermore, to grow the length of the candidate itemsets, combinations are used. All this inevitably makes the whole computation slow and could represent an issue when datasets are huge.

1.2 The Algorithm of Savasere, Omiecinski, and Navathe (SON)

The two pass MapReduce algorithm of Savasere, Omiecinski, and Navathe, known as the SON algorithm after the authors, constitutes an improvement on the simple randomized algorithm. The main idea is to divide the entire file of baskets into segments, called chunks, small enough that all frequent itemsets for the segment can be found in main memory. The candidate itemsets are those found frequent for at least one segment.

In the first pass through the data we read each chunk and processed it in parallel. In a second pass, we count all the candidate itemsets and select those that have support at least s as the frequent itemsets. This pass allows us to count all the candidates and find the exact collection of frequent itemsets. These two passes can be expressed as a MapReduce operation, summarized as follows:

1. First Map Function: takes the assigned subset of the baskets and find the itemsets frequent in the subset using the simple, randomized algorithm.

2. First Reduce Function: each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces those keys (itemsets) that appear one or more times. The output is the candidate itemsets.

3. Second Map Function: each Map task counts the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. The output is a set of key-value pairs (C, v), where C is one of the candidate sets and v is the support for that itemset among the baskets that were input to this Map task.

4. Second Reduce Function: takes the itemsets they are given as keys and sum the associated values. Those itemsets whose sum of values is at least s are frequent in the whole dataset.

1.3 FP growth Algorithm

The Frequent Pattern Growth is another association-rule mining algorithm. It overcomes the mentioned disadvantages of the A-priori algorithm by representing a faster solution. Its process is based on a Trie Data Structure.

How the algorithm works? First of all, items are gathered into baskets. Then, a minimum support, namely the frequency an item appears in the whole dataset, must be chosen by the user. It defines the lower bound of the “frequent pattern” set. The frequent pattern set contains all the frequent items sorted in descending order. After that, starting from the most frequent items, a new ordered item set is generated for any basket.

Here is where the Trie Data Structure comes into play. Starting from the first basket, all the items are put in a sequence following the order of occurrence in their ordered-item set. For the first basket the support count for each item is initialised as 1. One basket after the other, nodes are updated. If there are no direct links between subsequent items then a new branch is generated.

Next, for each item, a conditional pattern base is computed. It corresponds to the branches in the Trie Data Structure and they are stored as item sets. For each frequent item, the set of items that is common in all the paths in the conditional pattern base is taken and its support corresponds to the count of all the paths.

Ultimately, the frequent pattern rules are generated by pairing the items with all the possible combinations given by the frequent elements shared in the paths. So, items in the pairs are said to be bound by an association rule.

2 Pre-processing techniques

2.1 Text mining

As we are working with a massive dataset, the first step to pre-process the data is to import it as a Spark dataframe, which allows parallel computation and multiple-nodes data partition.

Since we need to convert articles into baskets to get the most frequent itemsets (and articles are strings of text), some of the most famous natural language processing techniques were applied using Spark NLP. It is an open-source state-of-the-art NLP library built natively on Apache Spark and released by John Snow Labs.

First of all, articles were transformed into documents, then sentences inside articles were detected and subsequently tokens obtained to extract single words out of each sentence. Next, tokens corresponding to punctuation and stop words were removed, and finally each of the words were lemmatized. All these processes were embedded into a pipeline which made possible the transformation of the dataset. Once the baskets were obtained, we needed to remove duplicates from them to work with sets inside baskets that are represented by unique items, or tokens in our case.

After completion of these steps, everything was set to proceed with sampling and the implementation of the algorithms.

2.2 Sampling

For the application of the algorithms we considered just fractions of the entire dataset, otherwise the computation would have lasted too long.

For the A-Priori and SON algorithms, a portion s equal to the 0.5% of the dataset was considered. The articles were chosen at random. Considering the algorithm was computed on singletons only, results were obtained in around 2 hours 15 minutes by using the Google Colab Notebook with 12Gb of pre-assigned RAM. The resulting sample was made of around 5,000 articles.

For the FP growth algorithm, the sample contained just 5% of the original data. This guaranteed to obtain results in less than 40 minutes. The resulting sample counted 50,342 articles.

3 The experiments

3.1 A-Priori Algorithm

We implemented the A-Priori algorithm from scratch. There is no built-in package available indeed, and for the implementation we made use of some transformations available in the Spark environment. As previously mentioned, since the computation would require a lot of time, we based our experiment on a sample equal to the 0.5% of the entire dataset.

The first step was about the transformation of the basket contained in the sample into a Resilient Distributed Dataset (RDD). This way the elements present in the sample were copied and distributed over different machines where tasks could be performed in parallel. Parallelization guarantees a faster and more efficient computation.

Next, we flattened the RDD so that all the items were extracted from the baskets and gathered under one array of strings. This made possible to obtain each item support by applying Map and Reduce transformations. Then, items having a support lower than a certain threshold were filtered out by using the filter method.

The threshold was set equal to 50, that is about 1% of the sample size, so that only a few items could be considered frequent after the computation. Normally the threshold is set as low as possible and this is done to include a reasonable number of items as “frequent”. The items could be discarded in further steps, and only combinations of items that show a high frequency in the dataset are returned by the algorithm.

However, due to time computational constraints, we focused our study on the most frequent singletons.

These were the results about the sample:

```
+------+----+
|Item  |Freq|
+------+----+
|work  |107 |
|year  |73  |
|school|65  |
|start |61  |
+------+----+
```

The confidence is a measure to evaluate the probability of having a couple of items inside a basket, given the presence of one of the two items. In this specific case, the aim was to calculate it for all the items in the frequent itemset. A bug in the code did not make this possible. In particular, a function to detect pairs among baskets in the sample was created, but an error regarding the data structure of the RDD was returned.

We could only infer that "work", "year", "school" and "start" were the most present words in the articles.

3.2 The Algorithm of Savasere, Omiecinski, and Navathe (SON)

We also implemented the SON algorithm, on the basis of the functioning of the A-Priori. As the first thing done, we reduced the number of partitions from the default value (45) to 10 by using the method *coalesce()*. This was done because the algorithm has a call to the *collect()* method which is very expensive in terms of memory.

Then, given the new number of partitions, an adjusted threshold corresponding to the minimum support was calculated. This time an item needed to be present in a partition at least 5 times to be considered as "frequent". Through Map, Reduce and filter operations a frequent itemset were obtained for each partition, and each time added to an originally empty RDD. The candidate itemsets were returned as a result of the first pass of the algorithm.

In the second pass, the support for the candidate items was calculated by means of a Reduce operation. A filter was later manually applied to filter out infrequent items. This operation should lead to the avoidance of false positives. Surprisingly, no items had a greater support than 1 meaning each of them were present in at most one partition. So a list of "frequent" (or maybe better say "not-very-frequent") items including various words was returned. Again, a bug in the code occurred. In our opinion, this may be due to the distribution of items among the different partitions. Since Spark is an efficient framework but requires long times to dig very deeper into errors, we were not able to come to a better solution.

3.3 FP growth Algorithm

Finally, we tested the FP growth algorithm. To analyse the results obtained by executing this method, we run two experiments. At first, we set the minimum support and the minimum confidence equal to 0.006. In fact, given that the sample accounted for a little more than 50,000 articles, 0.006 corresponded to a minimum support of around 300 articles.

The support is the percentage of baskets in which item X and Y are present together and it is calculated as $\frac{freq(X,Y)}{N}$. The confidence is the probability that a basket containing the item X (or a set of items) also contains the item Y and it is calculated as $\frac{freq(X,Y)}{freq(X)}$.

Among the most frequent items, the algorithm returned words like "year", "make", "state", "game", but also like "mayor", "hospital", "election", even though the latter were less frequent. The most frequent pairs of items above the threshold were represented by the tuples (play, game), (louis, st), (make, year) and (year, ago).

(louis, st) and (year, ago) were the couples of items with the highest confidence. It was 0.84 and 0.59 respectively, indeed, meaning that when the words "louis" and "ago" appeared inside an article they were very often accompanied by "st" and "year" respectively. The tuple (st, louis) had a confidence equal to 0.42 instead. This has a logic explanation. In fact, the word "st" came often together with the name of some other saint (like Jude, or Andres).

We also used another metric to evaluate the results: the lift. Lift is defined as the support of the union of X and Y over the support for X times the support for Y, in formula $\frac{Supp(X \cup Y)}{Supp(X) \times Supp(Y)}$, and is the probability of having Y in the basket with the knowledge of X being present over the probability of having Y in the basket without any knowledge about the presence of X.

High lift values means there is high dependency between the items. This was the case of the words "st" and "louis" (lift=46.7), while other logic pairs like "game" and "season" had a much lower value (lift=6.17). It means that a dependency were evident, but words could sometimes appear alone inside different baskets. A lift value equal to 1 stands for independence and just in the case of the tuple (make, year) the results got close to the benchmark with a value equal to 1.35. Yet, their independence was subtle.

The algorithm provided by the PySpark library also offered the possibility to infer predictions based on the association rules previously mentioned and the item sets associated with each article.

In some cases, we found a clear self-explanations of the results. For example, in an article where the phrase “play a role” was used the word “game” resulted from the prediction. In other cases, either the prediction was related to a large item set and the association was weak, or no prediction at all was returned.

To question the results, we run a second experiment. The minimum support and the minimum confidence were both set to 0.003. By lowering the thresholds, we wanted to include more items in the Frequent Pattern set. Frequent items increased from 436 to 1055 indeed.

This time, other couples of items resulted in having a high confidence, like “vice” and “president”, “united” and “state”, “ohio” and “state”. Apparently, compound words are well detected by this metric. The same couples of items resulted in having a high lift value. Intuitevely, items came together in the articles.

On the other hand, “state” and “make” were detected as independent words, having both a lift value equal to 0.94. And this makes sense too.

Finally, predictions were again generated but in this case predicted items were definitely a lot in most of the articles, giving no reasonable results. This lead us to consider the idea of adding some items, like “make”, to the stop words dictionary.

4 Ways to optimize scalability

Although Spark provides very good functionalities for processing data on a large scale, it requires awareness of some important information to scale up solutions with data size.

For instance, we realized that instructions to retrieve data and visualize them may be very demanding in terms of memory resources. This may be an issue, especially when the memory size is limited. In the worst case, the driver may suffer an OutOfMemory error which causes the runtime to crash and the work gets lost. So, it is important to reason over this kind of calls and to limit them. For the same reason, it is also fundamental to write efficient queries.

Choosing the best data structures is beneficial as well. It helps saving memory too. For instance, numeric columns should preferred to string columns. It is preferable to associate string values with encoded values, to work with them, and just retrieve information in the original format once they are needed back.

Caching is another efficient good technique to scale up solutions with data size. It is used to cache the intermediate results of transformations so that other transformation runs on top of cached will perform faster. For example, we cached results obtained through the first Map operation in the A-Priori algorithm to later perform the Reduce operation on top of that.

Using partitions and distributed dataframes is also highly recommended. However, the number of partitions should not be too big, otherwise the risk is to incur in high concurrency, namely multiple tasks are assigned to different executors without due consideration of memory usage.

5 Conclusions

In conclusion, we investigated some techniques to carry out market-basket analysis with the goal to discover frequent itemsets in a massive dataset. To apply the different algorithms, we had to obtain the baskets using some text mining techniques, tokenizing each sentence and obtaining lists of tokens representing unique words in sentences.

The A-Priori and SON algorithms did not give back satisfactory results, and this was unfortunately due to code inefficiencies. Anyway, we could notice that their implementation is quite slow, considered the resources provided by the Google Colab workspace, but also the reduced size of the sample.

The FP growth algorithms gave back better results instead. We could analyze the association rules provided by the installed package. The code took a reasonable amount of time to execute with respect to a bigger sample than the previous implementation. It must be said that there were no inefficiencies in the code. Confidence, support and lift made us possible to discuss about the properties of the most frequent itemsets. For instance, frequent items are (play, game), (louis, st), (year, ago), (state), (people).

In our opinion, the most frequent items did not lead to any extraordinary finding. However, the algorithm could suit well other text-based datasets, either they would refer to newspapers again, or for example to transactions carried out in supermarkets.

Furthermore, in our experiments, we realized that Spark requires to deal with its technical features in order to be able to scale up solutions with data size. The technical features in question are data structures in Spark, caching, partitioning and code efficiency, for example.

The code to reproduce this analysis can be found in the following GitHub repository.