Market Basket Analysis

Course of Algorithms for Massive Datasets

Giulia Scarpa 26780A

September 2023

1 Introduction

The market-basket analysis is used to describe a common form of many-many relationship between two kinds of objects: items and baskets (usually called 'transactions'). Each basket consists of a set of items (an itemset), and usually the number of items in a basket is much smaller than the total number of items[1]. The purpose of this project is to effettuate this analysis through the A-priori algorithm, implements from scratch. The project described in this document refer to the Yelp dataset, which is published on Kaggle[1], a website that provides lots of datasets for data science researches, and the project is developed in the Google colab environment, a free platform made available from Google.

2 Dataset

The Yelp dataset is released under the CC-BY-SA 4.0 license, with attribution required. This dataset is a subset of Yelp's businesses, reviews, and user data. In the most recent dataset you will find information about businesses across 8 metropolitan areas in the USA and Canada. Each file is composed of a single object type, one JSON-object per-line:

- **business.json:** contains business data including location data, attributes, and categories.
- **review.json:** contains full review text data including the user_id that wrote the review and the business_id the review is written for.
- **user.json:** user data including the user's friend mapping and all the meta-data associated with the user.
- **checkin.json:** checkins on a business.
- **tip.json:** tips written by a user on a business. Tips are shorter than reviews and tend to convey quick suggestions[2].

[1] https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset

Specifically, in this research we concentrate on the review.json file which is composed by:

- **review_id:** string, 22 character unique review id.
- **user_id:** string, 22 character unique user id, maps to the user in user.json
- **business_id:** string, 22 character business id, maps business in business.json
- **stars:** integer, star rating.
- **date:** string, date formatted YYYY-MM-DD.
- **text:** string, the review itself.
- **useful:** integer, number of useful votes received.
- **funny:** integer, number of funny votes received.
- **cool:** integer, number of cool votes received.

First of all I download the dataset in a pyspark Dataframe from the kaggle website to obtain the data to do the analysis described below. I use the text of the reviews (taken from the review.json file) for the market-basket analysis where I consider the text field as baskets and words, as items. I computed the number of row of the dataset, and it results that there are about 7 millions of rows. Because of the huge amount of dataset I decided to use just a small sample (1000 rows) to experiment the analysis that is transform in a RDD structure.

3 Data Pre-processing

Before the analysis it is strictly necessary to pre-process the reviews text to be able to work in a correct way. I describe below the steps I did for the pre-processing phase.

1. First of all I import the necessary libraries for the pre-processing phase. I use some libraries make accessible from nltk[2] to work with human language data.
2. The first step of pre-processing is to clean the reviews data from characters that don't correspond to an alphabetic char, make the text in a lower mode and tokenize the review to obtain list of words. I did it through the *word_tokenize* function. After this step we obtain the reviews text divided in tokens, it means that we don't have a unique text anymore for each review, but a sort of list of words for each review.

[2] https://www.nltk.org/

3. The second step is to remove the stopwords (words that are frequent but not relevant for the meaning of the text and they usually are ignore). To do that I loaded the stopwords from nltk library, considering the english language. Afterwards I created a set of these stopwords and after that I removed the stopwords contained in the lists of the reviews obtained in the previous step.

4. The third step is the lemmatization (the process to transform words in their canonic way, e.g. 'is' becomes 'be'). I created a function for the lemmatization using the nltk library and I filtered the the lists of words through the lemmatization to obtained the final pre-processed text review.

5. The last step is to check that in each text review there aren't duplicates. I did that transforming the tokenize list as a set to remove possible duplicates. At the end I return this set with the form of a list for every review.

4 Apriori Algorithm

4.1 Association rules: Support, Confidence and Interest

A set of items that appears in many baskets is said to be "frequent". We assume there is a number s, called the *support threshold*. If I is a set of items, the support for I is the number of baskets B_i for which I is a subset. We say I is frequent if its support is s or more.

$$support(I) = \{i : I \subseteq B_i\}$$

Frequent sets of items is often presented as a collection of if-then rules, called *association rules*. The form of an association rule is I $\rightarrow$ j, where I is a set of items and j is an item. The implication of this association rule is that if all of the items in I appear in some basket, then j is "likely" to appear in that basket as well[1].

4.2 Implementation

The A-Priori Algorithm is designed to reduce the number of pairs that must be counted, at the expense of performing two passes over data, rather than one pass. I decided to use just a sample of the dataset (1000 rows) because I found computational limits with the use of Google colab to experiment with the entire dataset.

Frequent items

I create a function that computes frequent single items and take in input a relative threshold and the dataset in the form of an RDD. I use the function FlatMap to reduce the RDD in a single list where then with the Map function

I associate the counting 1 to all the words. I reduce words with the function ReduceByKey, which reduces duplicates by increasing every time I find that specific word the count value that I associated via the map function executed previously. At the end I filter the frequent words using the filter function, where if a word has the counting value major or equal to the support threshold is consider to be frequent, on the contrary this function remove the word because it's not frequent. To conclude the function return the RDD list of frequent single items with their counting values and print the total time of the execution of the function.

Frequent pair itemsets

I create a function that computes frequent pair itemsets and take in input a relative threshold, the list of frequent single items and the dataset in the form of an RDD. First of all I map the list of frequent items to obtain just the words and not their counting values. In a second phase I pass in input to a broadcast function the collection of this list. This step is important to have better performances and to make the code more scalable and efficient. Then I can compute the couple of itemsets. I use in the original dataset the FlatMap function to compute the 2-combination of words doing a check in the list of frequent single items passed from the broadcast function. In this way I filter a part of the possible combinations, because I'm just using the words that are considered frequent. Afterwards with the Map function I associate the counting 1 to all the couple of words. I reduce these pairs with the function ReduceByKey, which reduces duplicates by increasing every time I find that specific couple the count value that I associated via the map function executed previously. At the end I filter the frequent itemsets using the filter function, where if a couple has the counting value major or equal to the support threshold is consider to be frequent, on the contrary this function remove the couple because it's not frequent. To conclude the function return the RDD list of frequent pair itemsets and print the total time of the execution of the function.

Final Apriori implementation

The final implementation is created through the use of the code described previously. I first of all compute the frequent single items, then I create a while cicle where I use the code described to compute the frequent pair itemsets, but in this case the difference is that I use a variable to compute not just the combinations of 2 items, but also the next combinations that they iteratively increment until the end of the algorithm that happens when there aren't others frequent itemsets to compute. To conclude the function return the RDD list of frequent items and itemsets and print the total time of the execution of the function.

Experiments

I experimented the apriori implementation with three different relative threshold: 0.03, 0.05, 0.07, while the final minimum support threshold was computed

with the product between the relative thresholds and the number of rows (1000). They computed respectively 716, 236 and 113 frequent items/itemsets. The total time for the computation was 604, 163 and 100 seconds.

4.3 Scalability

The project was developed through the using of the pyspark-3.4.1 version. The code is developed using the pyspark Dataframe and RDD structures to make the code scalable with the huge amount of data.

4.4 FPGrowth Algorithm

The FP-growth algorithm, where "FP" stands for frequent pattern, given a dataset of transactions, the first step of FP-growth is to calculate item frequencies and identify frequent items. Different from Apriori-like algorithms designed for the same purpose, the second step of FP-growth uses a suffix tree (FP-tree) structure to encode transactions without generating candidate sets explicitly, which are usually expensive to generate. After the second step, the frequent itemsets can be extracted from the FP-tree. In spark.mllib, they implemented a parallel version of FP-growth called PFP, PFP distributes the work of growing FP-trees based on the suffixes of transactions, and hence is more scalable than a single-machine implementation. The FPGrowthModel provides frequent itemsets in the format of a DataFrame with the following columns:

- **items:** (array type) A given itemset.
- **freq:** (long type) A count of how many times this itemset was seen, given the configured model parameters[3].

In the case of my project I use a min support of 0.05 and the FPGrowth fits the 1000-rows sample in in 94.7 seconds and computes 335 frequent itemsets.

5 Discussion

I can conclude saying that:

- More is the minimum threshold less is the time of computation of the algorithm because it filters more, consequentially the algorithm has to compute less combinations.
- This Apriori algorithm made from scratch it could be improve to have better performances because I found some difficulties with the time of computations when I tried to develop it.
- The FPGrowth is for sure faster than the Apriori algorithm, in half of the time compared to the apriori it calculates the frequent itemsets.

References

[1] Jeff Ullman Jure Leskovec Anand Rajaraman. In: *Mining of Massive Datasets*. Vol. 3. Cambridge University Press, 2014, pp. 201–229.

[2] Yelp Dataset. In: $https://www.yelp.com/dataset/documentation/main$.

[3] Apache Spark. In: $https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html$.

Declaration

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.