2017
2018

Data Mining Project

APRIORI ALGORITHM

RODI SCHEELE S1013847

ABSTRACT

The main goal of this project is to see differences in performance of different implementations of the Apriori algorithm. The algorithm itself is not hard to understand, the challenge lies within optimizing it in such a way that it runs fast.

For this project four versions of the algorithm were created and implemented. Two of those are based on a more classical approach, two of them are based on a paper[3] about an improved approach. The algorithms are applied to the Ta Feng[4] dataset, which is at its maximum about 120.000 records long.

The improved version is about 67% faster and therefore implemented successfully. The data itself doesn't mean that much, because they are mostly numbers.

CONTENTS

1. INTRODUCTION

This paper was written for the Data Mining course at Radboud University in. It mostly focuses on the differences in performance between the classical approach to the Apriori algorithm and a suggested improved implementation, both of which are implemented on a real life data set. The project consists of this paper and a repository[5]. The paper contains a description of the algorithm with some of its mathematics, some related works of the algorithm, a description of the code that was used, the results yielded by the code and concludes with an analysis of those results.

2. RELATED WORKS

2.1 In general

Finding frequent items and sets of items is a widely used concept in all sorts of applications. The Apriori algorithm has a lot of documentation behind it all over the internet with many implementations differentiating in their approach. A classical implementation is described in “Fast Algorithms for Mining Association Rules”[1] by Rakesh Agrawal and Ramakrishnan Srikant.

Most of the code used in this project has been inspired by the code written by nalinaksh[2], for which an adapted script was used during the course. A few blocks of code, specifically for reading the data files and transactions into a list, have been modified and re-used in the project. The script written by nalinaksh has been implemented in the project as well.

2.2 Improved Apriori Algorithm

The paper used for the improved algorithm is called “An Improved Apriori Algorithm for Association Rules”[3] by Mohammet Al-Maolegi and Bassam Arkok.

The paper starts by explaining the problems with the classic algorithm, mostly covering its bad performance on big datasets due to how costly scanning all the transactions is. It proposes an alternate approach to reduce the time spent searching transactions and looking for frequent item sets.

Their approach is to not only remember frequent items, but to remember the transactions those items occur in as well. This means that each frequent itemset is saved together with a list of transaction occurrences. The process behind this is explained in chapter three.

The paper only covers finding frequent item sets. However, if their approach is found to be successful in reducing the time required in finding frequent item sets, it should be possible to apply the same principle with finding other association rules.

The result of the experiment mentioned in their paper is that the improved version of the algorithm requires significantly less time to run, the average time reduction for them was 67%. It would be interesting to see if this project can achieve the same results.

3. THE ALGORITHM

Apriori is an algorithm for finding frequent item sets and association rules. Its main application is market basket analysis, meaning that it scans transactions looking for items and sets of items with a high frequency. The main problem with the algorithm is that it is rather slow, especially on large volumes of data. A classic implementation of the algorithm is shown in figure 2.1 as pseudocode.

```
k = 1
T = list_of_transactions
Lk = []
for i in T: // Generate L1 itemsets
    sup[i] = (count[i in T] / [count(T])
    if i > support:
        Lk.append(i)
while True: // Generate Lk itemsets where k > 1
    k+=1
    Ck = list(itertools.combinations(Lk, k))
    for c in Ck:
        s = 0
        for t in T:
            if c.issubsetof(t) == True:
                s+=1
        s = (s / transactions * 100)
        if s > support:
            Lk.append(c)
    if Ck == []:
        break
```

Figure 3.1 - Pseudo code snippet of a classic implementation of Apriori

The algorithm starts by calculating the support for each item in the total amount of transactions. The support is calculated by dividing the amount of times an item was found within the transactional data by the total amount of transactions. If the support is higher than the minimum support threshold the item is kept in the L_k, where in this case k = 1, frequent itemset.

The next step is to generate candidate item sets C_k of size k with the items from L_{k-1} for k > 1. Every candidate set C_k scans all transactions T looking for C_k as a subset of the transaction. The total amount of times a candidate set was found as a subset within T is divided by the total amount of transactions, which is the support rate for the candidate itemset. If the support rate is higher than the threshold it is added to frequent item sets for L_k.

This is repeated until all items in C_k have been scanned for subsets in T. Once this has been completed and all frequent item sets of size k have been found, this process is repeated for k+1 until no more candidate item sets C_k can be found.

This means that the amount of computations that must be done scales with the total amount of transactions. Each candidate itemset has to be looked up in the transactions, meaning that C_k is multiplied by T for every candidate. The time to find the first k=1 frequent item set is only T. The total amount of calculations can be described as a summation of $L_1(T)$ + *count*(C_k) * *count*(T) where *count*(C_k) * *count*(T) itself is a summation for each size k and every single candidate.

The biggest downside to approach is the fact that it is slow, especially on large datasets. This is due to scanning every single transaction for every single candidate item set, which is not an efficient way of finding frequent items.

This project implements two improved versions of this algorithm. The goal of those improved versions is to reduce the amount of calculations and time required to run the algorithm and find association rules.

The code snippet in figure 2.2 briefly describes the improved version of the algorithm based on the report by Mohammet Al-Maolegi and Bassam Arkok

```
k = 1
T = list_of_transactions
L1 = []
for i in T: // Calculate L1 and save transactions with the items
    sup[i] = (count[i in T] / [count(T])
    if i > supoprt:
        L1.append(i,[List_of_transactions_containing_i],support)
while True: // Use the saved transactions to calculate Ck and Lk
    k+=1
    L1 = []
    Ck = list(itertools.combinations(L1, k))
    for c in Ck:
        s = count(c.intersection(List_of_transactions_containing_i)) /
count(T)
        if s > support:
            Lk.append(c)
            for i in c:
                if i not in L1:
                    L1.append(i)
    if L1 == []:
        break
```

Figure 3.2 - Pseudo code for an improved implementation of Apriori

The main idea behind the improvement is that instead of only saving the ID's of items in frequent item sets, the transactions which contain the items are stored as well. This slightly increases the time required to calculate L1 because the transactions have to be stored as well, but it should reduce the time required for calculating Ck and Lk afterwards and should reduce time needed overall. As an example, each 1 itemset L_1 would look as follows

Item ID	Found in Transactions	Support
12300325416545131	15641533, 4583454, 4834153 ,7864361458, 8348354	8.8453147
48674315134863373	15641533, 4583454, 4834153, 4836435, 8348354 ,47864854	9.4486411
….	….	….

After finding all L_1 sets new candidate sets C_k are generated using the items from L_1. After generating new candidates C_k, the itemset containing all 1 item sets L_1 is emptied. The improved algorithm only looks for intersections within the stored transactions, instead of scanning all transactions and searching for subsets. The amount of intersections found divided by the total amount of transactions is the support rate for the candidate itemset.

If the support rate for the candidate itemset C_k exceeds the threshold, the candidate set is divided into single items again. Those single items are checked against the new L_1 (that has been emptied after generating C_k), if it doesn't contain the item already it adds it to the new single itemset. The new itemset for L_1 is used for generating new candidates C_k of size k+1, and the process of searching for intersections between transactions is repeated until no more items in L_1 remain.

The amount of calculations can be described as a summation of C_k * (*list_of_transactions_containing_i* ^ *every_item_in_ck*) plus the initial calculations for generating the first L_1. This approach should require less calculations than scanning the entire dataset for every candidate.

$L_1(T) + C_k$ * (*list_of_transactions_containing_i* ^ *every_item_in_ck*)

The downside to this approach is that it may take up more memory because more data must be stored while running the algorithm. However, the amount of extra memory used is not much and should not be a problem with modern machines.

4. THE PROJECT AND IT'S CODE

The project contains of four implementations of the Apriori algorithm. Three of those implementations have, for the most part, been written from scratch by me. The fourth implementation is based on a script written by nalinaksh[2] to check to see whether my personal implementation of the classic algorithm yields the same results and as a comparison of a true and optimized classic implementation versus the improved implementation.

The algorithms are applied to the Ta Feng dataset[4]. The entire project takes many hours to run due to the size of the dataset. The biggest possible dataset is 119578 transactions long, which contains transactions over the course of four months. Running this dataset on the classic nalinaksh algorithm takes about an hour and about fifteen to thirty minutes with the improved algorithm, depending on the machine it is ran on.

Something to note is that the support is hardcoded for item sets k > 1 and set to 10*k. This is done because I had found very few frequent itemset rules and thus didn't have a large sample size for finding frequent item sets, this made comparing performance between the algorithms quite tough. Hardcoding this to 10*k made the algorithm find more frequent sets, making it easier to see differences in performance.

The project contains comments in the code, explaining what is happening. The code itself could do with some more optimisation. The project can be found in the following repository

https://bitbucket.org/Azaiko/data_mining_project

The project consists of 6 scripts. Each of them is briefly described below.

The apriori.py script is the script that should be executed for finding the datasets. It takes about 4 hours to run the entire thing, for “quick” results I would suggest only running the 5000, 15000 and 29901 variants, this should take at most an hour. The 119578 version takes about 15 minutes to execute with the improved script and about 1 hour to execute with the classic script, depending on the machine.

The transformdata.py script transforms the data from the Ta Feng dataset to be suitable for the algorithms. The dataset itself consists of sales where each record is a sale of a single item. Because the dataset doesn’t contain transaction ID’s but it does contain dates and customer ID’s I have made the assumption that all items bought by one customer ID on a single day is one transaction. The output .txt files do still contain items with spaces and brackets, I wasn’t able to remove those with code and decided that it was not worth spending too much time on, since it is not the focus of the project. I used the replace all tool in notepad to remove whitespaces, and brackets [and].

The classicAprioriNalinaksh.py script is a modified version of nalinaskh his script that was used in the course. Finding the frequent 1-itemsets has been replaced by my own function so that it is consistent with the other implementations, since finding frequent 1-itemsets is not part of the improvement. This way all algorithms take roughly the same amount of time to generate 1-itemsets. The function used to generate new candidates and find new item sets is almost identical to the one used during the course.

The classicAprioriRodi.py script is my own implementation of the classical Apriori algorithm. It works slightly different to the one written by nalinaksh in that it first generates all candidates with all items from the previous item sets and then prunes them, rather than generating new item sets using previous item sets directly. It is a simpler and less optimized version of nalinaskh his script.

The improvedApriori.py script is my implementation of the improved version of the Apriori algorithm as described in the paper “*An Improved Apriori Algorithm For Association Rules”*. Finding the frequent 1-itemsets is consistent with the classic scripts. The difference in implementation lies within finding the larger item sets.

The improvedAprioriAdaptation.py is my personal adaptation of the improved version. The original improved version doesn’t include any frequent itemset pruning. This one does try to prune any frequent item sets that do not contain a subset of size k-1 before searching for intersections.

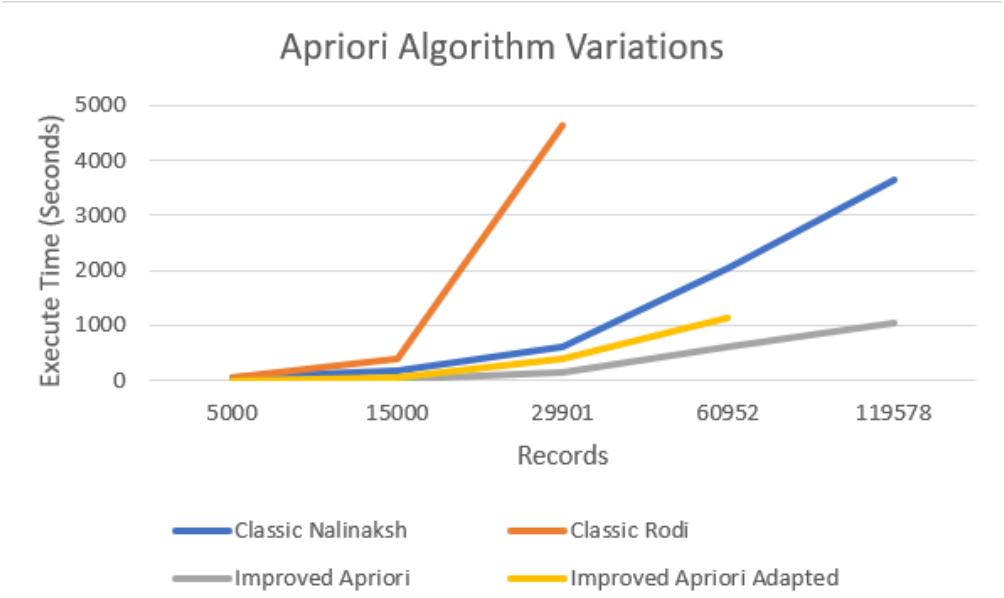
5. RESULTS

The results of this project are similar to the results documented in the article on which the improved algorithm is based upon. The original approach is significantly slower than the improved approach. The version with pruned candidate item sets before intersecting is actually slower than the version where the candidates are not pruned before intersecting.

It is important to note that the outcome for all the algorithms on one itemset yields the same results, therefore it is safe to assume that they all work fine and that they all do the same thing. The following table shows the amount of time in seconds it took for each script to run.

	Classic Nalinaksh	Classic Rodi	Improved Apriori	Improved Apriori Adapted
5000	46	56	6	7
15000	182	406	36	51
29901	620	4633	145	392
60952	2026	na	610	1133
119578	3643	na	1039	na

This data can be displayed as a graph. Showing that the improved version is much faster, by about 67 percent.



As stated before, all algorithms yield the same top 10 item sets. The frequent item sets differ depending on the size of the dataset. Below is a sample of the frequent 1 item sets with their support rates.

Itemset 5000	**Itemset 60952**	**Itemset 29901**
ID=4710265849066L Supp=16.52	ID=4714981010038L Supp=7.44	ID=4719090900065L Supp=5.49
ID=4710515537026L Supp=3.34	ID=4719090900065L Supp=3.1	ID=4710265849066L Supp=3.07
ID=4713071811159L Supp=3.26	ID=4710265849066L Supp=2.91	ID=4712425010712L Supp=2.83
ID=4710189820851L Supp=2.88	ID=4711271000014L Supp=1.99	ID=4714981010038L Supp=2.41
ID=4710036003581L Supp=2.34	ID=4710036003581L Supp=1.86	ID=4712425010255L Supp=2.07
ID=4714981010038L Supp=2.18	ID=4710114128038L Supp=1.86	ID=4710011401128L Supp=2.06
ID=4711080010112L Supp=2.16	ID=4711080010112L Supp=1.81	ID=4710018004605L Supp=2.03
ID=4710054380619L Supp=2.06	ID=4712425010712L Supp=1.67	ID=4710085120628L Supp=1.97
ID=20332433L Supp=1.98	ID=4710088410139L Supp=1.62	ID=4711080010112L Supp=1.9
ID=4710011401128L Supp=1.92	ID=4713985863121L Supp=1.51	ID=4710174053691L Supp=1.9
Itemset 15000	**Itemset 119578**	
ID=4710265849066L Supp=5.8	ID= 4714981010038L Supp=4.68	
ID=4712425010712L Supp=3.51	ID= 4711271000014L Supp=3.8	
ID=4719090900065L Supp=2.62	ID=4714981010038L Supp=2.41	
ID=4714981010038L Supp=2.05	ID= 4710114128038L Supp=1.58	
ID=4710011401128L Supp=1.93	ID= 4711080010112L Supp=1.55	
ID=20332433L Supp=1.87	ID= 4719090900065L Supp=1.42	
ID=4710054380619L Supp=1.76	ID= 4710088410139L Supp=1.36	
ID=4713985863121L Supp=1.75	ID= 4713985863121L Supp=1.33	
ID=4711080010112L Supp=1.75	ID=4711271000014L Supp=1.32	
ID=4710085120628L Supp=1.63	ID= 4710583996008L Supp=1.24	

The following is a sample of a top 10 frequent itemset for 2 and 3 items, where the support rate is the total amount of transactions where it was found rather than divided by total amount of transactions.

-------TOP 10 (or less) FREQUENT 3-ITEMSET----------
set= { 4710011401128L, 4710011401135L, 4710011405133L }, sup= 17
set= { 4710011401128L, 4710011401135L, 4710011409056L }, sup= 17
set= { 4710189820851L, 4710189851282L, 4710265849066L }, sup= 15
set= { 4710011401128L, 4710011401135L, 4710011406123L }, sup= 15
set= { 4710011401128L, 4710011405133L, 4710011409056L }, sup= 15
set= { 4710011401128L, 4710011401135L, 4710011401142L }, sup= 12
-------TOP 10 (or less) FREQUENT 2-ITEMSET----------
set= { 4710189820851L, 4710265849066L }, sup= 45
set= { 4710011401128L, 4710011401135L }, sup= 42
set= { 4710085120703L, 4710085120710L }, sup= 41
set= { 4710189820851L, 4710189851282L }, sup= 39
set= { 4710011401128L, 4710011405133L }, sup= 38
set= { 4710515537026L, 4713071811159L }, sup= 33
set= { 4710265849066L, 4710964103315L }, sup= 26
set= { 4710036000832L, 4710036000849L }, sup= 26
set= { 4710189851282L, 4710265849066L }, sup= 25

6. ANALYSIS AND EVALUATION

Looking at the results, the most notable thing is that the improved version of the algorithm is much faster. Another notable thing is that my implementation of the classic algorithm is much slower than the nalinaksh implementation. The final thing to note is that my attempt at speeding up the improved algorithm has only made it slower.

I had expected that my own implementation of the classic algorithm was going to be slower than the nalinaksh implementation. However, I did not expect it to be this much slower. My algorithm uses all items that are left in the L_1 item set to create C_k item sets and removes candidates that don't contain a subset of L_{k-1}, rather than using the previous item sets to generate new candidate sets. It does seem that this approach is not viable on large volumes of data.

The improved algorithm on the other hand is much faster than the nalinaksh algorithm. The paper "An Improved Apriori Algorithm for Association Rules"[3] stated that it should be about 66% faster, which is in line with our results. It shows that my implementation of the improved version is indeed faster, as described in the paper.

My attempt to increase the performance of the improved version by pruning candidate item sets was not a success. As is seen in the results, it takes longer to execute the program. It should be possible to optimize the improved version further by creating new candidates with the method used by nalinaksh, rather than my attempt. I however was not able to achieve this myself within the time window of the project. I think that the idea behind pruning candidates before searching for intersections should reduce execute time even further, if implemented differently and correctly.

All four algorithms produce the same top-10 results. Because of this it is safe to assume that all implementations perform as they should, with the difference laying in performance.

Looking at the results themselves, all database sizes produce similar rules for the top 10. This is expected, because as the sample size increases, the support should stay about the same with some possible minor differences. The biggest difference is with item 4710265849066L, which spikes at first and starts to stagnate with bigger sample sizes. This is most likely due to the smaller sample size. A plausible explanation for this would be that the item was on sale for a period of time and extremely popular, thus having a lot of sales within 5000 transactions.

The top 10 frequent 3-item sets contain a few items from the top 1 and 2 item sets. These results make sense because if an item is often bought, the chances of it being bought together with another item and creating a frequent 2- or 3-item set will be high as well.

7. CONCLUSION

The results are mostly as expected. The proposed improvement did indeed make the algorithm run much faster. The project and the paper did only touch finding support rules, however, the same principle can be applied to finding other rules as well.

The Apriori algorithm is quite simple to understand, the real challenge lies within implementing it in such a way that it takes as few calculations as possible. This is an algorithm that needs to be applied to large data sets, so optimization important here.

The code used for the project is not clean and tidy. Most of the code could do with some cleaning and optimization to further improve performance. Cleaning up the code should not change the results, however.

The results from the algorithm on the dataset are as expected as well. Nothing really springs to mind looking at them, this is partly due to the fact that all support rules are just numbers and therefore don't actually tell us that much. If it was possible to find names for the item ID's the results would probably be more interesting to analyse and look at.

8. LITERATURE

[1] – R. Agrawal, R. Srikant - *Fast Algorithms for Mining Association Rules (1994)* - http://www.vldb.org/conf/1994/P487.PDF

[2] – nalinaksh - *Association Rule Mining in Python (2016)* - *https://github.com/nalinaksh/Association-Rule-Mining-Python*

[3] – M. Al-Maolegi, B. Arkok – *An Improved Apriori Algorithm For Association Rules (2014)* - https://arxiv.org/ftp/arxiv/papers/1403/1403.3948.pdf

[4] – Ta Feng Dataset - https://stackoverflow.com/questions/25014904/download-link-for-ta-feng-grocery-dataset

[5] – Code repository - https://bitbucket.org/Azaiko/data_mining_project