Pragmatic Approach to Association Rule Learning in Real-World Scenarios

Håkan Sundell  
Department of Information Technology  
University of Borås  
Borås, Sweden  
Email: hakan.sundell@hb.se

Rikard König  
Department of Information Technology  
University of Borås  
Borås, Sweden  
Email: rikard.konig@hb.se

Ulf Johansson  
Department of Information Technology  
University of Borås  
Borås, Sweden  
Email: ulf.johansson@hb.se

Abstract—We present a pragmatic approach for designing an efficient tool for extracting knowledge from customer data in the retail industry, e.g., market basket analysis. Association rule learning is an established topic within data mining and knowledge discovery with a large interest from the business intelligence community. With a focus on properties from a real-world environment and with an aim to get customer insights on a cross-hierarchy level, we have chosen to build upon the common Apriori algorithm. This algorithm has been optimized for the chosen real-world environment and adapted for implementation on commonly available computing platforms and workstations using the Microsoft .net framework. Several parallelization strategies have been developed and experimental results indicate that a significant speed-up is possible and that the tool can be utilized for producing valuable information.

Keywords—Data Science.

I. INTRODUCTION

The industry is increasingly interested in utilizing available IT-infrastructure in order to know their customers and market better, within the common terminology of business intelligence. The first measure taken in order to increase the customer insight is usually market basket analysis, by analyzing the relatively information-rich data stored in the increasingly large and verbose transaction logs. One possible insight resulting from this analysis can be relations between bought items, represented as association rules.

Items in the retail industry are often divided into several levels of articles within a hierarchy as follows:

• Article, e.g. "Cucumber imported from Sweden".
• Group Article, e.g. "Cucumber".
• Merchandise Category Article, e.g. "Vegetables".

An example of a possible and likely known association rule between group articles is: \{Bread, Milk\} \rightarrow \{Butter\}

If also considering that the rule might differ for certain articles within a group in combination with other groups, a possible mixed and likely more interesting association rule, is: \{Bread, Milk 1\% fat\} \rightarrow \{Margarine 10\% fat\}

As a basis for association rule learning serves frequent itemset generation, which has been extensively researched in the literature. Mathematically, it is a straight-forward problem to solve, although with a relatively high time complexity, and hence a lot of research has been focused on finding efficient heuristics to reduce the computational load. One of the most popular approaches for this is the Apriori algorithm [1] by Agrawal and Srikant from 1994. This algorithm’s heuristics is based on the observation that frequent itemsets of length \(n\) must be subsets of frequent itemsets of shorter length \(n - 1\). The algorithm requires several passes over the transaction data set. Zaki has presented the Eclat algorithm [2] that reduces the number of required passes over the data set. Han et al. have presented the FP-growth algorithm [3] that takes a quite different approach, by instead constructing a complete graph over all frequent itemsets.

Tools for business intelligence should produce useful results. Accordingly to Freitas [4] these results should be accurate, comprehensible, and interesting. The first two properties are typically fulfilled by available tools, although the level of interest is a subjective measure and requires a domain expert.

There are several commercial tools available for market basket analysis. Many are part of large systems for business administration, and are often not accessible to small and medium-sized businesses (SME). There are also several open-source solutions available, although they are mostly designed for a relatively expensive client-server hardware architecture. One the other hand, there do exist free tools that can run stand-alone on workstations, e.g., Weka [5] which is a very popular and free tool suitable for experimenting with data mining algorithms. However, when using these tools with real-world data, it often appears to be "big data", e.g. the transaction data gets far too big for being able to handle in a reasonable time or not even possible to handle at all.

In this paper, we aim for an optimization of algorithms for cross-hierarchy association rule learning and designing of a computing- and memory-efficient tool for analysis of real-world transaction data. Moreover, it should be suitable for SMEs and running on contemporary workstation hardware, thus alleviating the big data problem.

II. ASSOCIATION RULE LEARNING

Let \(I = \{i_1, i_2, \ldots, i_d\}\) be the set of all items in a market basket data and \(T = \{t_1, t_2, \ldots, t_N\}\) be the set of all transactions. Each transaction \(t_j\) contains a subset of items chosen from \(I\). A transaction \(t_j\) is said to contain an itemset \(X\) if \(X\) is a subset of \(t_j\). Mathematically, the support count,
\(\sigma(X)\), for an itemset \(X\) can be stated as follows:

\[
\sigma(X) = \{|t_i|X \subseteq t_i, t_i \in T\| \tag{1}
\]

By using the identified itemsets and their respective support count, we can construct the corresponding association rules. In order to help judging which of the association rules that are of interest, a number of measures have been defined. Here, we use three of the most commonly used, e.g. support, confidence and lift, defined as follows:

\[
\text{Support, } s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N} \tag{2}
\]

\[
\text{Confidence, } c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \tag{3}
\]

\[
\text{Lift} = \frac{c(X \rightarrow Y)}{s(Y)} \tag{4}
\]

The Apriori algorithm [1] can be utilized to find frequent itemsets and is defined as follows in Algorithm 1.

**Algorithm 1** The original Apriori algorithm.

1. \(k = 1\)
2. \(F_k = \{i \mid i \in I \land \sigma(\{i\}) \geq N \times \text{minsup}\}\)
3. repeat
4. \(k = k + 1\)
5. \(C_k = \text{apriori-gen}(F_{k-1})\)
6. for each transaction \(t \in T\) do
7. \(C_t = \text{subset}(C_k, t)\)
8. for each candidate itemset \(c \in C_t\) do
9. \(\sigma(c) = \sigma(c) + 1\)
10. end for
11. \(F_k = \{c \mid c \in C_k \land \sigma(c) \geq N \times \text{minsup}\}\)
12. until \(F_k = \emptyset\)
13. Result = \(\bigcup F_k\)

The overall process of finding and constructing association rules from transaction data, can be described as the following steps:

- **Frequent Itemset Generation**, i.e., finding sets of items that have a minimum support, e.g. \(\{X,Y\}\).
- **Rule Generation**, i.e., out of the found frequent itemsets constructing relevant rules by using the corresponding permutations and order them by measures as confidence and lift, e.g. \(X \rightarrow Y, Y \rightarrow X\).

III. APPROACH

We are aiming for enabling the finding and construction of cross-hierarchy association rules. With cross-hierarchy we consider, in the first hand, associations that range over the two levels "group articles" and "articles".

We are also aiming for designing an efficient tool that can be effective in finding interesting rules in a real-world environment and accessible to smaller and medium-sized businesses. The real-world environment that we primarily have in mind, is the grocery retail market, e.g. supermarkets. Besides efficiency regarding computation and memory requirements, and as the tool needs to be semi-automatic in the respect that a domain expert needs to steer what is "interesting", a powerful and intuitive graphical interface is needed. Therefore we aim for an architecture suitable for stand-alone workstations and the Microsoft .net development platform using the C# language.

A. Modified Algorithm

We start by identifying some important problems with the original Apriori algorithm in a real-world implementation context.

- \(C_k\) can be very large as we have over 40 000 items! For example, \(C_4\) would be containing over \(2 \times 10^{18}\) combinations!
- Finding \(C_t = \text{subset}(C_k, t)\) is also a complex operation, basically it needs the same amount of computational efforts as finding all combinations of \(k\) items in \(t\).

Our approach is to instead merge the calculations of \(C_k\) and \(\text{subset}(C_k, t)\) into one single operation, see Algorithm 2.

**Algorithm 2** The modified Apriori algorithm.

1. \(k = 1\)
2. \(F_k = \{i \mid i \in I \land \sigma(\{i\}) \geq N \times \text{minsup}\}\)
3. repeat
4. \(k = k + 1\)
5. for each transaction \(t \in T\) do
6. \(C_t = \text{apriori-gen-subset}(F_{k-1}, t)\)
7. for each candidate itemset \(c \in C_t\) do
8. \(\sigma(c) = \sigma(c) + 1\)
9. end for
10. end for
11. \(F_k = \{c \mid c \in C_k \land \sigma(c) \geq N \times \text{minsup}\}\)
12. until \(F_k = \emptyset\)
13. Result = \(\bigcup F_k\)

B. Data representation

Individual articles and group articles are represented by their id’s, which is commonly a number. While calculating \(\sigma(c)\) for each candidate itemset \(c\), the count associated with each itemset needs to be stored in an efficient data structure. The dictionary abstract data type comes naturally for this need. The Dictionary class supported in Microsoft .net appears to be able to both efficiently store and find associations. We choose to represent itemsets (the key) as strings, e.g. "\{1, 2, 3\}", and counts (the associated value) as integers. As the dictionary’s underlying construction is using hash-tables, individual counts can be found in \(O(1)\). Note, that for cross-hierarchy purposes it is important to use different ids for articles and group articles, respectively.

In the work of implementing the previously optimized Apriori algorithm in C#, it can be observed that there is no need to compute and store the whole set \(C_t\) before increasing \(\sigma(c)\) for each \(c \in C_t\). Instead \(\sigma(c)\) is increased when each \(c\) is computed in \(C_t\). Thus, the simplified implementation of steps 6 to 9 of Algorithm 2 becomes as follows in Program 1.
C. Multi-threading

Contemporary workstations have normally access to parallel execution of programs. At least, some level of multi-core computing is available, and thus multi-threading should be exploited in the implementation as much as possible. It is apparent that the nested loops in Algorithm 2 have inherent data parallelism, although the proposed implementation in Program 1 is sequential. Within the Microsoft .net framework the Task Parallel Library [6], [7] offers several techniques for exploiting data parallelism.

We have designed two approaches for exploiting the inherent data parallelism using the task-based parallel language constructs:

- Parallelism over Permutations.
- Parallelism over Transactions.

1) Parallelism over Permutations: In this approach for parallelization, we focus on the nested loops of Program 1 that creates the permutations over each basket for the candidate itemset generation. As the task-based parallel engine in Microsoft .net has a known and large overhead, each task needs to be large enough to compensate for the corresponding overhead, and thus we only employ the Parallel.FOR on loops corresponding to 3 items or more. To avoid the need for a thread-safe dictionary, each thread has a local list. These local lists contain the increments to $\sigma(c)$ that are deferred to be performed afterwards in a sequential step. The parallel implementation over permutations is presented in Program 2.

2) Parallelism over Transactions: In this approach, we focus on the iterations over transactions in steps 5 to 10 in Algorithm 2. The aim is to create larger tasks than what can be achieved over the parallel permutation generation for each transaction. We first make sure that transactions are read from the dataset in large chunks and employ the Parallel.FOREACH on 1000 transactions at a time. In order to properly balance the size of the tasks towards the computer’s actual number of computing cores, the Partitioner class is used. The parallel incrementing of $\sigma(c)$ is handled by the thread-safe ConcurrentDictionary class. The resulting implementation is shown in Program 3.

---

Program 1

```csharp
for (int i = 0; i < (keys.Count - 1); i++)
{
    long key1 = keys[i];
    if (dictionaryLevel1.ContainsKey(key1))
    {
        for (int j = i + 1; j < keys.Count; j++)
        {
            long key2 = keys[j];
            if (dictionaryLevel2.ContainsKey(key2))
            {
                string keyName = key1 + "," + key2;
                if (dictionaryLevel2.ContainsKey(keyName))
                    dictionaryLevel2[keyName]++;
                else
                    dictionaryLevel2.Add(keyName, 1);
            }
        }
    }
}
```

Program 2

```csharp
A parallel implementation of the candidate itemset generation with their respective counting, for each transaction, based on exploiting the inherent data parallelism in the permutation loops per transaction.

```csharp
for (int i = 0; i < (keys.Count - 1); i++)
{
    keys.Count - 1; j++)
    {
        long key2 = keys[j];
        if (dictionaryLevel2.ContainsKey(key2))
        {
            string keyName = key1 + "," + key2;
            dictionaryLevel2[keyName]++;
            for (int k = j + 1; k < keys.Count; k++)
            {
                long key3 = keys[k];
                if (dictionaryLevel3.ContainsKey(key1 + "," + key3))
                {
                    string keyName = key1 + "," + key2 + "," + key3;
                    dictionaryLevel3[keyName]++;
                    if (dictionaryLevel3.ContainsKey(keyName))
                        dictionaryLevel3[keyName]++;
                    else
                        dictionaryLevel3.Add(keyName, 1);
                }
            }
        }
    }
}
```

Program 3

```csharp
A parallel implementation of the candidate itemset generation with their respective counting, for each transaction, based on exploiting the inherent data parallelism in the iterations over transactions.

```csharp
for (int t = range.Item1; t < range.Item2; t++)
{
    List<long> keys = transactions[t].keys;
    if (keys.Count > 1)
    {
        for (int i = 0; i < (keys.Count - 1); i++)
        {
            long key1 = keys[i];
            if (dictionaryLevel1.ContainsKey(key1))
            {
                for (int j = i + 1; j < keys.Count; j++)
                {
                    long key2 = keys[j];
                    if (dictionaryLevel2.ContainsKey(key2))
                    {
                        string keyName = key1 + "," + key2;
                        dictionaryLevel2[keyName]++;
                        if (dictionaryLevel2.ContainsKey(keyName))
                            dictionaryLevel2[keyName]++;
                        else
                            dictionaryLevel2.Add(keyName, 1);
                    }
                }
            }
        }
    }
```
D. Articles and Group Articles

Normally, each $k$-itemset $c \in C_t$ is generated by $\text{apriori-gen-subset}(F_{k-1}, t)$ by simply enumerating all possible combinations of items in each transaction $t$ and then pruning those combinations not feasible. As the order of the items within each itemset is not of importance, the number of possible combinations before pruning is $n^k_k = n! / (n-k)!$ where $n$ is the number of items in $t$.

Now we want to also find itemsets that represents relations between articles $A$ and group articles $G$. We have identified two approaches for cross-hierarchy frequent itemset generation in relation to Algorithm 2:

1) Let the set of items in each transaction $t_i$ contain both articles and group articles. In essence it means that each transaction $t_i = \{i_1, i_2, \ldots, i_x\}$ will be treated internally for frequent itemset generation as $\{a_1, a_2, \ldots, a_x, g_1, g_2, \ldots, g_y\}$ where $a_j \in A$ and $g_j \in G$. Thus the numbers of possible combinations before pruning is $(2n)^k_k = (2n)! / (2n-k)!$.

2) Let the set of items in each itemset $X$ contain both articles and group articles. In essence it means that each $c \in C_t$ for each transaction $t$ can be seen as $\{a_1, a_2, \ldots, a_x, g_1, g_2, \ldots, g_y\}$ where $x + y = k$. For each possible combination of $x$ and $y$ the number of possible combinations before pruning is $n^k_k = n! / (n-k)!$.

The number of possible combinations of $x$ and $y$ is $k+1$.

It is apparent that approach 1 will be more computationally expensive than approach 2 for larger values of $k$. For example for $k = 5$ we have that approach 1 will have $2^5 = 32$ times more possible combinations to explore and approach 2 will have only $5 + 1 = 6$ times more possible combinations than would be required if were to only consider pure articles as items. Approach 1 also requires more complex pruning criteria to avoid nonsense relations between articles and the group article that adhere to the very same item in the transaction.

E. Browsing

When the association rules are generated from the correspondingly found frequent itemsets, it is important for the domain expert to be able to efficiently browse the relatively huge set of rules. Hence, mechanisms are needed for effectively reducing the selection of rules to view, and browse. Here the measures Support, Confidence and Lift from Equations 2, 3 and 4 are of direct interest. These have been implemented as graphical sliders, see Figure 1, and respond directly to a new computation of the current view of rules for browsing. In order for the graphical interface to remain responsive, the actual computation is handled by a background thread.

Moreover, the user is also supplied with other means to reduce the view of association rules, as for example selection boxes where a subset of group articles of interest can be selected. To alleviate the process of finding interesting rules, we have also supported the user with a special button that can be used to directly remove some certain group article or article of interest from the current view - thus reducing the scope in efficient steps, quickly narrowing down to the interesting rule.

IV. EVALUATION

We have performed an experimental study to examine the behaviour and characteristics of our implementations under different conditions. As a basis we are using real-world data from actual transactions performed by customers in a supermarket during summer 2014. Some key characteristics of this retail business and the corresponding transactions are that the whole line of products contains 59 600 articles and 438 group articles.

Our experiments are divided into two main types; firstly several benchmarks utilizing a subset of the dataset and secondly a real-world case using the whole dataset.
Table 1
EXPERIMENTAL RESULTS FROM BENCHMARKS ON DUAL-SOCKET WORKSTATION WITH INTEL XEON E5-2690. THE RESULTS SHOW THE CORRESPONDING NUMBER OF FREQUENT ITEMSETS, MEMORY CONSUMPTION, AND EXECUTION TIMES FOR THE THREE RESPECTIVE IMPLEMENTATIONS.

<table>
<thead>
<tr>
<th>Pruning Support</th>
<th>Transactions (nr)</th>
<th>Frequent Itemsets (nr)</th>
<th>Memory (bytes)</th>
<th>Sequential (ms)</th>
<th>Parallel P. (ms)</th>
<th>Parallel T. (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>20000</td>
<td>3498</td>
<td>7457808</td>
<td>12316</td>
<td>9431</td>
<td>5997</td>
</tr>
<tr>
<td>0.01</td>
<td>40000</td>
<td>3822</td>
<td>8009592</td>
<td>26609</td>
<td>18812</td>
<td>11397</td>
</tr>
<tr>
<td>0.01</td>
<td>60000</td>
<td>3763</td>
<td>8011976</td>
<td>38049</td>
<td>28105</td>
<td>17011</td>
</tr>
<tr>
<td>0.01</td>
<td>80000</td>
<td>3740</td>
<td>8001232</td>
<td>50393</td>
<td>37251</td>
<td>22563</td>
</tr>
<tr>
<td>0.005</td>
<td>100000</td>
<td>4041</td>
<td>8038168</td>
<td>66300</td>
<td>48024</td>
<td>29218</td>
</tr>
<tr>
<td>0.005</td>
<td>60000</td>
<td>19968</td>
<td>9954576</td>
<td>72097</td>
<td>43593</td>
<td>26708</td>
</tr>
<tr>
<td>0.005</td>
<td>80000</td>
<td>19532</td>
<td>10172016</td>
<td>95032</td>
<td>57712</td>
<td>34609</td>
</tr>
<tr>
<td>0.005</td>
<td>100000</td>
<td>19341</td>
<td>10078600</td>
<td>11397</td>
<td>34609</td>
<td>26708</td>
</tr>
<tr>
<td>0.005</td>
<td>120000</td>
<td>20913</td>
<td>10303448</td>
<td>139852</td>
<td>26708</td>
<td>34609</td>
</tr>
<tr>
<td>0.005</td>
<td>140000</td>
<td>20314</td>
<td>10503448</td>
<td>160560</td>
<td>45555</td>
<td>30566</td>
</tr>
<tr>
<td>0.005</td>
<td>160000</td>
<td>20000</td>
<td>10703448</td>
<td>181850</td>
<td>678017</td>
<td>45555</td>
</tr>
</tbody>
</table>

A. Benchmarks

The defined micro-benchmarks serve for answering questions about time efficiency and memory consumption of the different implementations, as well as direct properties of the
transaction dataset in regard to frequent itemset generation. The input variables that are changed, are the minimum support value for pruning and the number of transactions processed. The experiments have been performed on a workstation with dual-socket Intel Xeon E5-2690 with totally 16 cores and 64 GB of memory running Windows Server 2012. All measured values are presented in Table I.

Figure 2 shows the relation between support pruning and the numbers of itemsets generated for our selected transaction dataset, with support pruning values ranging between 0.01 and 0.0005. As can be seen, the number of generated itemsets increase exponentially with decreasing values for support pruning.

Figure 3 shows how memory requirements for the itemset generation phase increase with the number of generated itemsets. Apparently this relation is linear, and can be expressed approximately as 7000000 + 120 * n where n is the number of itemsets.

Figure 4 shows the time needed for the various implementations with regards to different support pruning values ranging between 0.01 and 0.0005, when analyzing a total of 100 000 transactions from the dataset. Apparently, the parallelism over permutation implementation can perform up to 50% faster than the sequential implementation, and the parallelism over transactions implementation can perform up to 40% faster than then parallelism over permutations implementation.

When selecting a minimum of support pruning of 0.0001, in less than one hour, 11 079 187 itemsets are generated. The corresponding number of association rules after pruning for confidence and lift are 40 682 206, with the total memory requirements of 13 GB. It is still possible to smoothly scroll over all those rules, and changes in the view filter using the sliders takes only some seconds to respond.

As we neither have access to any commercial system, nor are aware of any tool capable of cross-hierarchy association rule generation, we have not been able to directly compare with similar solutions. However, we have tried to open the dataset in Weka [5], although with failure as the tool stopped after some hour due to exceeding of memory requirements.

V. CONCLUSION

We have presented a new tool for association rule learning aimed for use within the retail industry, with the following properties:

- Finds rules spanning over multiple hierarchy levels.
- Based on the well-established Apriori algorithm.
- Optimized for usage within a real-world environment.
- Parallelized using several strategies of data parallelism.
- Implemented using Microsoft .net.
- Designed with a versatile and useful graphical interface.

We believe that our results will have interest within the research community as well as direct practical benefits in the retail industry, contributing to enhanced business intelligence.

ACKNOWLEDGMENT

This work was supported by the Swedish Retail and Wholesale Development Council through the project Innovative Business Intelligence Tools (2013:5)

REFERENCES