Condensed Representation of Frequent Itemsets

Daniel Serrano
daniel.serrano@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisboa, Portugal

Cláudia Antunes
claudia.antunes@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisboa, Portugal

September 2014

Abstract

One of the major problems in pattern mining is still the problem of pattern explosion, i.e., the large amounts of patterns produced by the mining algorithms when analyzing a database with a predefined minimum support threshold. The approach we take to overcome this problem aims for automatically inferring variables from the patterns found, in order to generalize those patterns by representing them in a compact way. We introduce the novel concept of meta patterns and present the RECAP algorithm. Meta patterns can take several forms and the sets of patterns can be grouped considering different criteria. These decisions come as a trade-off between expressiveness and compaction of the patterns. The proposed solution accomplishes good results in the tested dataset, reducing to less than half the amount of patterns found.

Keywords: Frequent itemset mining, pattern explosion, compaction, summarization

1. Introduction

Our ability to generate and collect data has been increasing rapidly [1] and the use of the World Wide Web has been one of the major catalysts for this. It is then essential to be able to extract relevant information from the huge amount of data that we have in our power. To achieve this goal, a great deal of research has been carried out in various fields of study, ranging from statistics to machine learning and database technology.

Data mining is, according to [2], the nontrivial extraction of implicit, previously unknown, and potentially useful information from data and thus it is a solution to the problem of analyzing and deriving meaning out of the data. Data mining comprises four demands: high-level language, meaning the discovered information should be presented in an easy-to-understand language for the human user; accuracy, as one must guarantee that the knowledge acquired by analyzing data correctly portrays the contents of the database; interesting results, which means the discovered knowledge should be considered interesting according to user-defined biases; and efficiency, making it possible for the user to discover relevant information in a reasonable amount of time.

In order to derive meaning from the data, one must find regularities in it so that it is possible to infer general cases or what is usually called patterns. A pattern is an expression in some language, describing a subset of the data or a model applicable to the subset [3]. Patterns are considered interesting when they are novel, useful and non-trivial to compute [2].

The problem of pattern mining was first proposed by [4] for market basket analysis. Consider a supermarket with a large collection of items. If the marketing team has enough information, by analyzing its customers previous transactions, it will be able to predict the kind of products customers buy in the same transaction. This will allow the marketing team for rearranging the items in a smarter way that will eventually lead to increases in sales. Frequent itemset mining, or just pattern mining, strives for completeness [5], i.e., discovering all patterns that satisfy certain conditions and although this is a useful goal, it has its drawbacks. A strong drawback is that typically many patterns will fulfill the constraints at hand in a given problem. This drawback is known as pattern explosion: pattern mining often produces an unwieldy number of patterns, comprising strongly redundant results which become prohibitively hard to use or understand.

Since the seminal paper on pattern mining [6], a lot of research is aimed at compacting the patterns found and some encouraging results have been achieved in the meantime, proving that there are techniques to better handle the amount of information we face nowadays.

In this paper, we present RECAP (REgression CompAct Patterns), a new pattern compaction algorithm. We aim to derive relations between the different attributes that are common to a set of patterns. This type of generalization allows us to ab-
extract a group of patterns into a new pattern, called a meta-pattern, which relies on a language natural to the user and more compact than the usual representation.

The rest of this paper is organized as follows. Section 2 presents relevant concepts in the area of pattern mining and provides insight to related work that in the past has tried to mitigate the problem of pattern explosion. Section 3 clearly states the goal of our proposal by formalizing the problem and explaining key concepts needed to understand the solution. Section 4 describes the algorithm that allows the creation of meta-patterns. In Section 7 we present experimental results and evaluate the overall performance and effectiveness of our approach, taking into consideration compression ratio, memory usage and computational time, among others. The paper concludes with some guidelines for future work.

2. Background

The problem of pattern mining can be described as follows. Let \( I = \{i_1, i_2, \ldots, i_n\} \) be a set of possible items (e.g., the items for sale in a store). An item is then a proposition – an attribute-value pair. For example, buying milk can be translated to milk=5 (attribute milk and value 5), stating that the client bought 5 units of milk or to milk=true (or just milk) stating that the client bought some milk. Let \( D \) be a set of database transactions where each transaction \( T \) is a particular set of items such that \( T \subseteq I \) (e.g., the items a client bought). A set of items is referred to as an itemset. The support of a given itemset \( IS \subseteq I \) in \( D \), denoted by \( sup(IS) \) is the percentage of transactions in which \( IS \) occurs; in this context, a pattern is a frequent itemset.

The problem of pattern mining is: given the minimum support threshold, \( \sigma \), determine all itemsets \( P \) such that \( sup(P) \geq \sigma \).

It is clear that the threshold value given as input to any pattern mining algorithm will be crucial in regards to what patterns are found. A large support value may yield only obvious patterns in the data, whilst very small values might be the ones that reveal the most interesting patterns that are not trivially graspable. The focus of our work comes from trying to mitigate the problem known as pattern explosion, i.e., the large amount of patterns produced by mining algorithms when restricted to low support values. Several consequences arise from this phenomenon: high memory requirements, difficulty in understanding the reported patterns and strongly redundant results.

There is a reasonable amount of work done in the area of pattern compression and particularly on trying to reduce the number of what will be considered the set of relevant patterns. One of the first methodologies used was that of constrained sets, whereby one can filter out important patterns that describe the patterns found (typically maximal [7] and closed sets [8], [9]) or patterns that are very similar to other patterns (-free sets [10]). More complex constraints can be used, taking into account disjunction relations [11] or the derivability (or inference) of a larger number of patterns from a particular subset [12].

A different compaction approach uses the concept of top-k patterns, which consists in limiting the number of patterns reported to only the most k important ones. This can specifically target the top k frequent closed itemsets with a minimum length [13], the k patterns that best approximate a collection of patterns found [14], the k most significant and redundancy-free set of patterns [15] or even organize them by relevance, considering metrics like F-score or \( \chi^2 \) [16].

Another potential solution comes from the use of the minimum description length principle [17]. In the context of pattern mining it is used to find the set of frequent itemsets that best compress the database at hand. The basic premise to this is that if an itemset is to be deemed as interesting, then it must yield a good (lossless) compression of the database.

Pattern compaction has also been seen as a clustering problem by separating frequent itemsets in well-defined groups of patterns. The compression task may then create a representative pattern able to represent the common regularities in each group. A greedy algorithm with an agreed tightness measure among patterns in a cluster has previously been used [18]. Alternatively, profile-based clustering which relies on the concept of a master pattern and distribution vectors has also been proposed [19].

Additionally, there has also been work on the efficient storing of patterns, particularly using an FP-tree-like structure, the CFP-Tree or Condensed Frequent Pattern tree, in which the main idea is to store only closed frequent itemsets [20].

From a completely different line of research, inductive logic programming [21], [22] also proposed important contributions to pattern discovery, since a large number of algorithms have been proposed in this area for allowing the inference of data from background knowledge. The Classic’cl system, influenced by various recent approaches, integrates a wide range of constraints such as minimum frequency, maximum generality, exclusive disjunctions and condensed representations [23].

All these approaches have some sort of cost: information might be lost and thus the end-user, interested in inferring the maximum amount of information he possibly can, might not find useful information in the discovered patterns; it can happen that the top-k patterns selected are not enough
for the end-user to get a grasp of all information that might be obtained with more discriminative representations; the efficient structures previously mentioned are not thought of to store a form of meta-patterns, i.e., these structures normally store patterns in their simplest form, but never complex representations like the ones presented next.

3. Problem Statement
Consider a database and one of the commonly used pattern mining algorithms (Apriori [6] or FP-Growth [24], for example). When restricted to low support values, these algorithms will report an unwieldy number of patterns, which makes them hard to understand since most of the times they are presented to the end-user as a list of patterns. Even if they are grouped using some criteria (e.g., group them by attributes), often times patterns will present some kind of regularities, i.e., redundancies between them, that might be overlooked by the user. For example, one might find that for a given set of patterns, the value of an attribute is always the same; or the value of an attribute has a coherent relation with some other attribute (e.g., one of the attributes takes values which are always twice the value of the other attribute). For this reason, we believe a compaction scheme could be set up in which such regularities are automatically identified and new patterns, called meta-patterns, are generated enabling a compact representation of the patterns found.

Definition 1. A meta-pattern is a compact representation of a set of patterns.

Consider the set of patterns in the toy example of Table 1. In this example, we initially have four patterns but these can be compacted in two meta-patterns, able for representing all the original patterns. To understand the process that leads to this outcome, we introduce the concept of meta-item.

Definition 2. A meta-pattern is composed of a non-empty set of meta-items. A meta-item is a compact representation of a set of items.

In some way, a meta-item plays the same role as a variable. As an example, consider again the data in Table 1. The first meta-pattern \( \{ \text{Chips} = 2, \text{Sodas} = \{4, 5\} \} \) is composed by two meta-items: one describing the values taken by attribute \( \text{Chips} \) (with \( \text{Chips} = 2 \)) and another one describing the values taken by attribute \( \text{Sodas} \) (with \( \text{Sodas} = \{4, 5\} \)). Different types of meta-items can be inferred, and we define three particular types of meta-items: constant meta-items, disjoint meta-items and regression meta-items.

Definition 3. A constant meta-item is a meta-item that describes a set of items sharing the same attribute-value pair.

In Table 1, the set of patterns that share attributes \( \text{Chips} \) and \( \text{Sodas} \) both contain item \( \text{Chips} = 2 \), which can lead us to the definition of a meta-item \( \text{Chips} = 2 \). A constant meta-item can be seen as an abstraction of a set of items that are equal among them.

Definition 4. A disjoint meta-item is a meta-item that describes a set of items sharing the same attribute but different values over the set of items.

Continuing from the previous example, one can see that \( \text{Sodas} \) takes values 4 or 5. We thus can infer a disjoint meta-item given by \( \text{Sodas} = \{4, 5\} \). It will guarantee that only one expression is used to represent this set of items, instead of two or more. In particular, this type of meta-item is necessary when considering that we can infer a more complex type of meta-item, the regression meta-item, which is also the most expressive one.

Definition 5. A regression meta-item is a meta-item that describes a set of items sharing different values for the same attribute, which in turn are dependent on another attribute through a simple linear regression model. A regression meta-item describes the values of a given attribute \( y \) (dependent variable) using a single explanatory attribute \( x \) (or target variable), a slope \( m \) and an intercept \( b \), such that \( y = mx + b \).

The second meta-pattern in Table 1 contains a case of such a type of meta-item, in which the values of attribute \( \text{Beers} \) are dependent on the values of attribute \( \text{Diapers} \), with a slope of 2 and an intercept of 0 (\( \text{Beers} = 2\text{Diapers} + 0 = 2\text{Diapers} \)).

We can define broader concepts on top of these notions, which will be useful to understand the algorithm presented in later sections of this document.

Definition 6. A constant meta-pattern is a meta-pattern consisting only of constant meta-items (e.g., \( \text{Diapers} = 2, \text{Beers} = 3 \)).

Definition 7. A disjoint meta-pattern is a meta-pattern consisting of one or more disjoint meta-items and may or may not contain constant meta-items (e.g., \( \text{Chips} = 2, \text{Sodas} = \{4, 5\} \)).

Definition 8. A regression meta-pattern is a meta-pattern containing at least one regression meta-item (e.g., \( \text{Diapers} = \{1, 2\}, \text{Beers} = 2\text{Diapers} \)).

Our problem is then to find regularities among a set of patterns and be able to extract expressions capable of describing relations among attributes. The goal is to decrease the amount of information presented to the user, while increasing its expressiveness and understanding.
4. RECAP Algorithm

The problem we aim at solving is that of automatic inference of the minimal set of meta-patterns that describe an entire set of patterns. The first step for this to happen is presented next. We must group the patterns in such a way that we only try to generalize sets of patterns that make sense to be grouped together under a singular explanatory expression a unique meta-pattern.

After that process, we introduce the meta-pattern generation algorithm, which will use the groups of patterns previously identified in order to infer the relationships between attributes in the identified groups.

4.1. Grouping phase

The reasonable heuristic way of performing the grouping of patterns is by placing patterns that share a set of attributes in the same group. Consider once again Table 1. The first set of original patterns only makes sense to be grouped together because they both share attributes Diapers and Beers. It would not make sense to try to infer the type of expression we defined in the previous section if we tried to group retrieved patterns that do not share the same set of attributes. For example, a generalization between Diapers = 1, Beers = 2 and Chips = 2, Sodas = 4 would not be successful.

Although this heuristic is a good starting point, it is arguable if we should group together every pattern that shares the same set of attributes. There can be cases where the support of one pattern (or more) is too different from the support of every other pattern in the group of that set of attributes. In that case, it might be better to create a separate group to consider significantly different support values between patterns sharing the same attributes. For that reason, this grouping of patterns is performed also taking into account the support of the patterns found. Furthermore, we define a distance measure as criteria to decide whether a pattern should be included in a given group of patterns.

The distance measure is calculated comparing the first computed support of a given group of patterns (which will be from here on referred to as reference support) and the support of a pattern identified as sharing the same set of attributes of that group. The formula for this measure is given by equation 1:

\[ d = \frac{|reference - support|}{reference} \]  

For each analysed pattern, the distance \( d \) between a pattern and the group of patterns that shares the same set of attributes with it cannot be larger than a user-defined threshold \( \Delta \sigma \), i.e., we must guarantee that \( d \leq \Delta \sigma \). The formula uses the reference support as the denominator so that this distance can be given as a relative percentage value.

The grouping phase is then the stage in which we prepare the patterns to be meta-mined, i.e., we group them considering the specific data model we are using. In the case of transactional records the behaviour will be that of the pseudo-code in Algorithm 1.

The grouping of patterns is done through a hash table (\texttt{groupedPatterns}), where each key corresponds to a unique identifier of the set of attributes shared by the group of patterns. Each of these identifiers contains a group of patterns that share the same attributes and have ”close” supports (considering the distance measure previously defined). Figure 1 depicts the data structure described.

The patterns returned by frequent itemset mining algorithms usually give items ids as they appear in the dataset, ignoring the attributes shared, so it is possible that for example Chips = 2 and Chips = 3 have completely different ids and appear at very distinct positions in the patterns discovered. For that reason, a unique identifier for each pattern (line 2) is created by concatenating the ordered strings that represent the items. The order is important because if we did not order the strings corresponding to the attributes of a given item, we would possibly generate distinct keys (and consequently different groups) for patterns sharing the same set of attributes.

The loop beginning in line 3 is only concerned with placing the pattern in the correct group. We begin by finding the hash table corresponding to the set of attributes in the pattern (line 3). We then look for the entry that is closest in support to the new pattern (line 4). If the closest group has a support value too far from the threshold \( \Delta \sigma \), then we create a new group (line 6). We then add the newly analyzed pattern to the group it fits best in (line 8).
Algorithm 1 Grouping phase algorithm pseudo-code.

Require: patterns, ∆σ
Ensure: list of groups of patterns grouped by attributes and support
1: for pattern in patterns do
2:   uniqKey ← generateUniqKey(pattern)
3:   groups ← groupedPatterns.get(uniqKey)
4:   group ← findClosest(pattern, groups)
5:   if distance(pattern, group) > ∆σ then
6:     group ← newGroup(pattern)
7:   end if
8:   group.add(pattern)
9: end for
10: return listGroups(groupedPatterns)

4.2. Meta-pattern generation
After the patterns are grouped together considering the heuristic described before (sharing attributes and closeness in terms of support values), we focus on the process of generalization.

For each group of patterns, we perform an analysis on the group size and on the number of attributes it comprises. The two trivial cases occur when either the group only contains one pattern (no generalization can be made) or the group contains more than one pattern but only one attribute. In the first case, we generate a meta pattern comprising a single constant meta item, for each item present in that pattern. The second case implies generating a meta pattern simply consisting of a disjoint meta item on that attribute.

The most complex case happens when there are multiple patterns in a group, each of which comprising more than one attribute. The process of finding regularities in a group of patterns is as follows. For each attribute in every pattern belonging to that group, we try, in an ordered fashion, to describe that attribute in terms of another attribute, providing it is consistent with all patterns in the group, then two cases might arise. If the slope is 0, then we create a constant meta item (\( y = 0x + b = b \)) to represent the dependent attribute. If the slope is different from 0, then a regression meta item on that attribute is created. In both cases, a disjoint meta item must be created in order to represent the possible values that the target attribute takes. The process of choosing the attributes to be generalized is repeated until we have tried to generalize all attributes that are not already classified as dependent or target ones.

The pseudo-code for this process is given in Algorithm 2. In the first loop (line 1), \( i \) represents the index corresponding to the attribute chosen to be the dependent variable in a regression run. If it has not yet been used as a target variable in some other regression computation, we try to generalize it (line 2). Simple linear regression is computed taking into account \( i \), the regression error and all the dependent attributes as constraints, since the algorithm needs to consider the \( i \)th attribute as the dependent variable, while not trying to generalize it using any of the variables already inferred as dependent. This second constraint guarantees that no indirectly dependent generalizations are created, i.e., cases in which an attribute can be described in terms of another attribute which is in turn dependent of a third attribute (e.g., Table 2). In the process of unfolding a meta-pattern (generating the corresponding patterns that meta-pattern is summarizing), this type of dependencies would hinder the task of inferring the values a given attribute takes. For example, to infer the value of Lettuces in the example of Table 2, we would need to get the value of Beers first and only then would we be able to calculate the value of Chips. By avoiding this, the process of inferring the values of a regression meta-item becomes simply calculating \( mx + b \) for each target value. Indirectly dependent generalizations may also confuse the end-user, since as it can be seen in Table 2 we would have attributes expressed in terms of different target variables. In that example, generalization could be made in terms of the same attribute (target variable Diapers), which simplifies the process of understanding dependencies.
Algorithm 2: Regression meta-patterns algorithm pseudo-code

Require: patternGroup, \( \varepsilon \)
Ensure: metaPattern

1: for \( i \leftarrow 1, \text{patternGroup.numAttributes} \) do
2:   if not independent\((i)\) and not target\((i)\) then
3:     regression.run(patternGroup, \( i \), dependents, \( \varepsilon \))
4:     if regression.successful then
5:       if regression.slope = 0 then
6:         metaPattern.add(constantMetaItem\((i)\))
7:       else
8:         setDependent\((i)\)
9:       end if
10:   metaPattern.add(regressionMetaItem\((i, \text{regression})\))
11: end if
12: else
13:   setIndependent\((i)\)
14: end if
15: end for
16: if allIndependent(patternGroup.attributes) then
17:   return null \( \triangleright \) failure
18: else
19:   for \( i \leftarrow 1, \text{patternGroup.numAttributes} \) do
20:     if independent\((i)\) or target\((i)\) then
21:       metaPattern.add(disjointMetaItem\((i)\))
22:     end if
23:   end for
24: end if
25: return metaPattern \( \triangleright \) success

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Indirect dependency example</th>
<th>Desired meta-pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diapers = 1, Beers = 2, Lettuces = 3</td>
<td>Diapers = {1, 2}, Beers = 2Diapers, \text{Lettuces} = 0.66\text{Beers}</td>
<td>\text{Lettuces} = 3\text{Diapers}</td>
</tr>
<tr>
<td>Diapers = 2, Beers = 4, Lettuces = 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Indirect dependency example.

The last loop in the function (line 20) exists so that disjoint meta-items are created out of the items which attributes were found to be independent (could not be generalized because no relationship between attributes could be inferred) or target attributes (were used as a target variable in a regression meta-item).

The regression meta-item is created considering the regression itself, because it contains the values for the slope and intercept. A regression run is considered to be successful if and only if the sum of mean squared errors is less than the user-specified error (represented by \( \varepsilon \)).

In the worst-case scenario, all attributes will be deemed as independent, i.e., we will not be able to find any regression meta-item in the group of patterns (lines 17 and 18). In that case, we will subsequently try to find patterns that have the same values between them and only vary in terms of one attribute, using what we call the Disjoint Model. In that case, we try to find patterns that have the same values between them and only vary in terms of one attribute. As an example consider the case of Table 3. The set of patterns in that group is not generalizable considering simple linear regression, but two groups can be further inferred if we split the group in regards to the "more decisive" attribute. The idea is similar to (the inverse of) information gain in decision trees, but here it is simplified by using only the relative frequency of items in the group of patterns.
5. Extension for Sequential Patterns

Our algorithm is flexible enough to work with other types of patterns, but this cannot go without testing and proper verification. What we aimed to do was apply our algorithm to a different data model. We applied it to a different type of patterns. Instead of transactional records, we meta-mined sequences. The sequential patterns at hand were the result of applying the TDPrefixGrowth sequence mining algorithm by [25] over time series. Time series, which are sequences of data points measured at consecutive equally spaced time intervals, are able to infer the evolution of a given attribute across time.

The TDPrefixGrowth tries to infer this type of behaviours in data. Specifically, it tries to identify evolutive trends (i.e., patterns that occur multiple times according to a specific periodic behaviour).

With this generalization, we were able to achieve compaction of this type of sequential patterns that reached about six times less information being presented to the end-user. Refer to [26] for more information and details about handling sequential patterns (including the explanation of different compaction strategies used).

6. Optimizations

One of the first things we noted when running the RECAP algorithm on transactional patterns, was the amount of disjoint meta-patterns that were found within the patterns returned by the mining task. If most of the times we are going to fail in using the Simple Linear Regression Model, which will lead to the use of disjoint relations between patterns, thus generating disjoint meta-patterns, then we might prepare the grouped patterns in some way that makes it easier for the process of finding these disjoint relations.

This process is enabled by the use of a custom data structure that is shaped like an n-ary tree. Details of its construction and traversal can be found in [26].

The second optimization we put in place is more general and it has to do with the way we are performing a crucial part of the RECAP algorithm, trying to find simple linear regression relations between different attributes.

By monitoring our application, we found was that most of the processing time was spent in I/O operations.

Initially, we were using the Weka library [27] in the standard way, i.e., we were reading information from arff files in order to build a data model consisting of a set of patterns to be analysed. This is the normal way Weka is used to process data which is static, i.e., does not change from one run to another. This methodology will eventually lead to a large amount of system calls to deal with the I/O operations of opening, writing, reading and closing the files at hand. For that reason we decided to dynamically create the data model for each set of patterns. Instead of generating new files and simply reading them, we had to generate the data model ourselves.

In the end we bypassed a great amount of the I/O overhead in which we previously incurred, thus improving the overall processing time.

7. Experimental Results

All the experiences were performed on a Pentium T4200 Dual Core 2.00 GHz machine with 4GB RAM running Windows 7. We used the implementation of Apriori algorithm available in the D2PM project [28] to collect the frequent itemsets that are further post-processed by our algorithm.

We used a private academic dataset to run the experiments. It contains the marks of students enrolled in the Computer Science and Engineering bachelors degree from 1990 to 2000 at Instituto Superior Tcnico, Universidade de Lisboa in Portugal. Each one of the 2614 entries represents a different student and his corresponding marks in twelve courses. The grading scale goes from 10 to 20, with 10 being the minimum mark a student must achieve to be considered approved to any course. Only positive results have been used. In order to maximize the probability of generalization, a normalization was performed on the marks, in which each pair of consecutive values is given the lower bound (records with values 10 or 11 become 10, records with values 12 or 13 become 12, etc.). This is true for every pair except the last one, which also includes mark 20 (values 18, 19 and 20 become 18), since the highest mark is very unlikely to happen. This can be seen as a rough approximation to the American grading system, here translated from letters to numeric values (necessary for our algorithm).

We run the algorithm with different support values and analyze it considering four metrics. The first is compression ratio in terms of patterns found, which will be computed by comparing the number

---

Table 3: Example of application of the Disjoint Model to a group of patterns.

<table>
<thead>
<tr>
<th>Original pattern group</th>
<th>Fine-grained pattern groups</th>
<th>Meta-patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diapers = 2, Beers = 3</td>
<td>Diapers = 2, Beers = 3</td>
<td>Diapers = 2, Beers = {3, 5}</td>
</tr>
<tr>
<td>Diapers = 2, Beers = 3</td>
<td>Diapers = 2, Beers = 3</td>
<td>Diapers = 4, Beers = 7</td>
</tr>
<tr>
<td>Diapers = 4, Beers = 7</td>
<td>Diapers = 4, Beers = 7</td>
<td>Diapers = 4, Beers = 7</td>
</tr>
</tbody>
</table>
of patterns reported by our algorithm with: 1) the number of patterns found by the Apriori algorithm alone; 2) the number of closed and maximal patterns. We also perform time and memory consumption tests. We are interested in understanding how space and time will vary with support variations. Another comparison we will make to Apriori has to do with time. Being a post-processing algorithm it is desirable that the amount of time spent in compaction is the minimum possible.

The results in terms of numbers of patterns reported to the end user (in the form of meta-patterns) from running the compaction algorithm with minimum support set to 10%, 7.5%, 5% and 2.5% are shown in Figure 2. Note that we allow permissive pattern grouping in the preparation phase, making virtually infinite. It is clear that compaction is always successful, i.e., we can always reduce the number of patterns reported in comparison to Apriori. The figure also shows that the ratio of compaction (ratio between the number of patterns before compaction and after it) increases with the decrease of support. This happens because the amount of patterns found by the Apriori algorithm increases in an exponential rate. In this way, a broader set of patterns will be able to be grouped together into a meta-pattern capable of describing that set of patterns, because the number of similar (hence, generalizable) patterns will also increase. We also run the Eclat algorithm [29] using the openly available implementation by [30] to assess the amount of patterns reported to the end user when using closed and maximal sets. As can be seen, the closed patterns take no effect in reducing the number of patterns found without missing values (note the overlap between patterns and closed). The maximal set is successful in reporting fewer patterns but not to the extent that meta-patterns do, with meta-patterns incurring in about half the number of patterns obtained by using the maximal set strategy.

In Figure 3, we compare the computational time spent by the compaction algorithm and by Apriori. It is clear that although being a post-processing algorithm, the amount of time spent in compaction is negligible when compared to the task of frequent itemset mining. It is again clear that with the decrease of support, the amount of time spent both during Apriori as well as in the compaction process increases (since with the decrease of support one will have more patterns to compact). However, the time spent on compaction grows much slower than the pattern mining step. When running the experiments in the education dataset, we registered the amount of patterns that contained regression meta items, the number of meta-patterns involving disjoint relations and the number of patterns that remained the same even after compaction. In Figure 4 we can see that with increasing support values, the ability to generalize patterns increases (due to the reasons mentioned when describing the first chart). In particular, when setting the minimum support at 10%, we verify that no simple linear regression models could be successfully created from the set of patterns found and about half the meta-patterns reported were exactly the same as the ones reported by Apriori. The majority of meta-patterns contain disjoint generalizations, nevertheless. It is interesting to see that the largest number of meta-patterns involving simple linear regressions is found when the minimum support is set to 7.5%, which does not represent a minimum or maximum value on the supports used. This is very dependent on the data itself and consequently on the patterns found with the frequent itemset mining algorithm.

As to memory usage (Figure 5), the results follow what was expected, i.e., an exponential increase of the amount of memory used with the decrease of minimum support. For the lowest minimum support, memory used reaches approximately 50 MB, which shall be explained by the use of expensive Java data structures like hash maps, array lists and tree sets. We believe that it is possible to improve efficiency of the algorithm regarding memory consumption by using cheaper data structures.
(e.g., simple arrays) in detriment of more complex abstractions, or ultimately creating custom data structures. In terms of time spent on actually running the compaction algorithm (Figure 6), besides the analysis performed on the overhead of it being a post-processing algorithm, we also report the time in seconds that the compaction task takes. When set to the lowest minimum support value the algorithm runs in less than 1.2 seconds.

8. Conclusions

The explosion of the number of patterns discovered by frequent itemset mining algorithms, along with the difficulty on analyzing such number of patterns, have impaired the generalized use of pattern mining in data analysis.

In this paper, we proposed a method for compacting the discovered patterns into a more manageable set, creating abstractions of the original patterns discovered the meta-patterns. Experimental results show that even considering very low levels of support, the compaction rate is high, and the time spent on this step is negligible compared to the time spent on pattern mining.

The proposal presented in this paper is a post-processing algorithm. A disadvantage of this approach is that the discovery of meta patterns has to be performed after Apriori or any other frequent itemset mining algorithm has found frequent itemsets. If the process of generalization was done in runtime, i.e., while analyzing the itemsets in the database, we are confident better performance times could be achieved. This, together with the use of a custom data structure especially targeted at finding the types of meta patterns we strive to find, could lead to better memory use.

Furthermore, the properties of the patterns grouped together are related to the concepts of additive and multiplicative models that are referenced in some biclustering algorithms [31] (particularly when considering simple linear regression relations). This is another point of focus that shall be further explored, especially when trying to infer regression meta items. For now, if two meta patterns with regression meta items can be inferred out of one group of patterns, this will not be found by our algorithm, since it tries to generalize attributes using the simple linear regression model considering the whole set of patterns in a group. If we have knowledge of clusters of patterns that share behavior on a subset of their attributes (hence, biclusters), we would allow for more (finer grained) meta patterns to be found.

Acknowledgements

This work is partially supported by FCT Fundação para a Ciência e a Tecnologia, under research project D2PM (PTDC/EIAEIA/110074/2009).

References


Jiawei Han, Jianyong Wang, Ying Lu, and Petre Tzvetkov. Mining top-k frequent closed patterns without minimum support. In IEEE International Conference on Data Mining. IEEE Computer Society Washington DC, USA, 2002.


Dong Xin, Hong Cheng, Xifeng Yan, and Jiawei Han. Extracting redundancy-aware top-k patterns. In Knowledge Discovery and Data Mining, Philadelphia, Pennsylvania, USA, 2006. ACM.


Dong Xin, Jiawei Han, Xifeng Yan, and Hong Cheng. Mining compressed frequent-pattern sets. In 31st VLDB Conference, Trondheim, Norway, 2005.

Xifeng Yan, Hong Cheng, Jiawei Han, and Dong Xin. Summarizing itemset patterns: A profile-based approach. In Knowledge Discovery and Data Mining, Chicago, Illinois, USA, 2005. ACM.


Luc De Raedt. Inductive logic programming, 2010.


Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 2000.

A. Barreto and C. Antunes. Finding periodic regularities on sequential data: Converging, diverging and cyclic patterns. In Submitted to 18th International Database Engineering & Applications Symposium, Porto, Portugal, 2014.


