Learning compact models for representing sets of patterns

Daniel José Domingos Serrano

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Professor Cláudia Martins Antunes

Examination Committee
Chairperson: Professor José Carlos Alves Pereira Monteiro
Supervisor: Professor Cláudia Martins Antunes
Member of the Committee: Professor Alexandre Paulo Lourenço Francisco

November 2014
Things get done only if the data we gather can inform and inspire those in a position to make difference.

Mike Schmoker
Acknowledgments

I would like to thank my advisor, Professor Cláudia Antunes, for her invaluable experience, expertise and guidance provided throughout the development of my thesis. Her patience and calm were always of the utmost importance. This journey was filled with moments of joy, confidence but also doubt. Her optimism and security kept me going and striving to accomplish the best possible version of what we initially set out to do.

I am grateful to my family, in particular to my parents José and Lurdes and my brother Bruno, who have always believed in me and always had something nice to say, even in the most troubling moments.

To my closest friends, some of them with whom I have shared most of my college life, people I have known for quite some time now, you have been the best friends I could never have asked for.

Last but not least, I would like to thank my colleagues in the research projects educare and D2PM, particularly António Barreto, Daniel Cardoso, Fábio Carballo and Andreia Silva, for all their valuable insights, fruitful discussions and critical thinking, as well as the laughs shared and support in moments of distress.

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

Um dos principais inconvenientes da descoberta de padrões ainda é o problema da explosão de padrões, i.e., o grande número de padrões produzidos pelos algoritmos de descoberta de padrões quando analisando uma base de dados com um valor mínimo de suporte. Várias técnicas, algoritmos e sistemas foram propostos de maneira a mitigar este problema, uns através de filtros aplicados aos padrões, outros através da compactação dos padrões encontrados, mas nenhum deles através da abstracção dos padrões encontrados em representações mais gerais baseadas em modelos matemáticos. Neste trabalho, propomos a noção de meta-padrão - uma abstracção de um conjunto de padrões que partilham características comuns, e um novo método para encontrar o grupo de meta-padrões capazes de representar um dado conjunto de padrões. A abordagem que tomamos procura inferir automaticamente variáveis a partir dos conjuntos de padrões encontrados, de maneira a generalizá-los e tornar possível uma representação compacta. O objectivo é preservar tanta informação quanto possível, usando um algoritmo eficiente e correspondente estrutura de dados que beneficie o processo de inferência de tais variáveis. Os resultados experimentais demonstram que é encontrado um significativo nível de compactação em termos dos padrões apresentados, dando assim ao utilizador resultados mais fáceis de entender.

Palavras-chave: Descoberta de padrões, explosão de padrões, compactação, sumarização, eficiência
Abstract

One of the major drawbacks in pattern mining is still the problem of pattern explosion, i.e., the large amount of patterns produced by the mining algorithms when analysing a database with a predefined minimum support threshold. Various techniques, algorithms and systems have been proposed to mitigate this problem, ones by filtering patterns, others by compacting the patterns found, but none of them by abstracting the patterns found in more general representations based on mathematical models. In this work we propose the notion of meta-pattern - an abstraction of a set of patterns that share common characteristics, and a new method to find the set of meta-patterns able to represent a given set of patterns. The approach we take aims for automatically inferring variables from sets of patterns found, in order to generalize them and allow for representing them in a compact way. The goal is to preserve as much information as possible, while using an efficient algorithm and corresponding data structure to benefit the process of inference of such variables. Experimental results show that our approach achieves significant pattern compaction ratios, thus providing the user with results that are easier to understand.

Keywords: Pattern mining, pattern explosion, compaction, summarization, efficiency
## Contents

Acknowledgments ................................................................. v
Resumo ................................................................. vii
Abstract ................................................................. ix
List of Tables ............................................................... xiii
List of Figures ............................................................... xvi

1 Introduction 1
   1.1 Goal and Contributions ............................................. 1
   1.2 Thesis Outline .................................................... 2

2 Background 3
   2.1 Closed Frequent Itemsets ........................................... 5
   2.2 Maximal Frequent Itemsets ......................................... 6

3 Literature Review 7
   3.1 Filtered Mining .................................................... 7
      3.1.1 Constrained Sets ............................................. 7
      3.1.2 Top-k Patterns ............................................... 11
      3.1.3 Approaches based on Minimum Description Length .......... 12
      3.1.4 Clustering-based Approaches ................................ 14
      3.1.5 ILP-based Approaches ........................................ 17
   3.2 Pattern Storage .................................................. 18
      3.2.1 FP-trees ..................................................... 18
      3.2.2 CFP-trees ................................................... 19
      3.2.3 Suffix Trees ............................................... 21

4 Meta-Patterns and RECAP Algorithm 23
   4.1 Problem Statement ................................................ 23
      4.1.1 Requirements ............................................... 25
   4.2 RECAP Algorithm ................................................ 27
      4.2.1 Grouping phase ............................................. 28
      4.2.2 Generation phase .......................................... 30
## List of Tables

2.1 Training examples and background knowledge necessary for learning the daughter relation.  3
2.2 Transactional database D for Closed Frequent Itemsets .......................... 5

3.1 Transactional database D for Non-Derivable Itemsets. .......................... 10
3.2 Supports of itemsets in D. ................................................................. 10
3.3 Tight bounds on the support of itemset ABCD. ................................. 11
3.4 Transactional database D for Minimum Description Length. ................. 13
3.5 Patterns and respective supports in a database D for greedy clustering. .... 14
3.6 Example transactional database. .......................................................... 19

4.1 Example of transactional patterns and corresponding meta-patterns. ......... 24
4.2 Meta-pattern generation for lossy representation and regression error. ....... 26
4.3 Constant meta-pattern generated for single pattern. .............................. 30
4.4 Disjoint meta-pattern generated for group of 1-itemset patterns. .............. 30
4.5 Indirect dependency example. ............................................................... 31
4.6 Example of application of the Simple Linear Regression Model to a group of patterns. ................................. 33
4.7 Group not generalizable using simple linear regression model and resulting finer-grained groups using disjoint model. ................................. 34
4.8 Example of application of the Disjoint Model to a group of patterns. ........ 35
4.9 Examples of sequential patterns found using the TDPrefixGrowth algorithm. 38
4.10 Examples of sequential meta-patterns. ................................................ 39
4.11 Patterns and corresponding meta-patterns with relaxation on the duration. .... 39
4.12 Patterns and corresponding meta-patterns when mining periods. .............. 40
4.13 Example of a group of patterns ............................................................ 41
# List of Figures

2.1 Itemset lattice of database D .................................................. 5

3.1 Standard code table for the database in Table 3.4, with associated cover and encoded
database ................................................................. 14

3.2 FP-tree for the example database. ............................................. 19

3.3 Database and corresponding CFP-tree ...................................... 20

3.4 Example of a suffix tree. ....................................................... 21

4.1 Overview of the RECAP algorithm ........................................... 28

4.2 Data structure used during the grouping phase. .......................... 30

4.3 Data structured created from the group of patterns in Table 4.13 .... 41

4.4 Main components of the optimized data structure. ....................... 42

4.5 VisualVM performance report. ................................................. 44

5.1 Information reported to the user when varying the grouping variable $\Delta \sigma$. ................................. 50

5.2 Comparison on the amount of information reported to the user for transactional records. ........................ 50

5.3 Compaction ratios considering different values on regression error. ................................. 51

5.4 Precision and recall for different minimum support values considering a fixed value for
regression error. ................................................................................................................. 52

5.5 False negatives, false positives and true positives Venn diagram. .... 53

5.6 Meta-pattern generation example. .............................................. 54

5.7 Precision and recall for minimum support values of 2.5% and 7.5%. .... 55

5.8 Types of meta-patterns found for different values of regression error. ................................. 55

5.9 Types of meta-patterns found for regression error $\varepsilon = 0$. .......... 56

5.10 Time and space performance of the RECAP algorithm over transactional patterns. ................................. 56

5.11 Overhead of RECAP over Apriori as a post-processing algorithm. ................................. 56

5.12 Comparison on the amount of information reported to the user for cyclic patterns over the
HHP dataset. ................................................................................................................. 58

5.13 Types of meta-patterns found for cyclic patterns over the HHP dataset. .... 59

5.14 Precision and recall considering different compaction strategies for cyclic patterns over
the HHP dataset. ................................................................................................................. 60
5.15 Experimental results for different cyclic compaction strategies with minimum support minsup = 1% over the EEG dataset. ......................................................... 61
5.16 Comparison on the amount of information reported to the user for converging patterns over the HHP dataset. .......................................................... 61
5.17 Types of meta-patterns and corresponding precision/recall for converging patterns over the HHP dataset (minsup = 1%). ....................................................... 62
5.18 Types of meta-patterns and precision/recall for converging patterns over the EEG dataset (minsup = 1%). .......................................................... 63
5.19 Comparison on the amount of information reported to the user for converging patterns over the EEG dataset (minsup = 1%). ....................................................... 63
5.20 Overhead of RECAP over TDPrefixGrowth as a post-processing algorithm (HHP dataset with fixed minsup = 1%). ....................................................... 64
5.21 Time consumption of the RECAP algorithm over sequential patterns for different compaction strategies (HHP dataset). ....................................................... 64
5.22 Memory usage over sequential patterns for different compaction strategies (HHP dataset). 65
5.23 Comparison between the non-optimized and optimized versions of the RECAP algorithm. 65
Chapter 1

Introduction

Our ability to generate and collect data has been increasing rapidly [Han and Kamber, 2006] 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 has played an important role in this area and it is defined, according to [Frawley et al., 1992], by the “nontrivial extraction of implicit, previously unknown, and potentially useful information from data”. The main goal of knowledge discovery and data mining is to discover hidden patterns, i.e., “expressions in some language, describing subsets of the data or models applicable to the subsets” [Fayyad et al., 1996]. Patterns are considered interesting when they are novel, useful and non-trivial to compute. In its essence, data mining is concerned with causal-effect detection. The hidden patterns that are found enable us to detect cause-effect relations between the elements under analysis. For example, patterns inferred from available records of customer transactions might reveal sets of items that are often bought together. The marketing team might then rearrange the way items are placed on the shelves eventually leading to increases in sales.

Frequent itemset mining, also referred to as pattern mining, has been one of the main processes of knowledge discovery. However, a practical inconvenience has been found as result of the mining task. Pattern mining often yields a large number of common, redundant and uninformative patterns. This problem is commonly known as pattern explosion and results in poor understanding of the hidden information obtained and its inadequate treatment to retrieve it to the end user.

1.1 Goal and Contributions

Since the seminal paper by [Agrawal et al., 1993], a great amount of effort in 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 [Lin and Kedem, 1997, Han et al., 2002, Siebes et al., 2006, Xin et al., 2005, Dzeroski, 1996]. However, the algorithms...
proposed thus far still suffer from some problems, including (i) lack of expressiveness; (ii) inherent information loss and (iii) efficiency concerns. In this paper, we present a new pattern compaction approach. We aim to derive relations between the different attributes that are part of a given set of patterns. This type of generalization allows us to abstract 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 work presented here is integrated, and uses, the Domain Driven Pattern Mining (D2PM) framework.

1.2 Thesis Outline

The rest of this thesis is organized as follows. In chapter 2 we review relevant concepts of pattern mining. Chapter 3 presents insights to related work in the area of pattern compaction. Chapter 4 clearly states the goals of our work and presents some of the requirements we must meet, as well as our final proposal. In Chapter 5 we present a case study in which we apply the algorithm to a dataset in a specific domain and compare different compaction strategies. The document concludes in chapter 6 with an overview of the contributions and final remarks.
Chapter 2

Background

Two domains of research have been connected to the causal effect detection. Historically, Inductive Logic Programming was the first area of interest that tried to solve the problem of identifying such relations of causality. It investigates the construction of first-order definite clause theories from examples and background knowledge. First-order logic is an extension of propositional logic, thus being richer, particularly allowing for the existence of constants, variables, functions and quantifiers.

In [Dzeroski, 1996], the author gives an overview of the inductive logic programming systems used at the time. The works described are important to understand the foundations of this knowledge discovery paradigm.

Suppose we want to define the relation \( \text{daughter}(X, Y) \), which states that a person \( X \) is a daughter of person \( Y \), in terms of the relations female and parent. If we consider the examples and background knowledge in Table 2.1, we conclude two positive and two negative examples are present. It is then possible to infer that the daughter relation is defined as follows:

\[
\text{daughter}(X, Y) \leftarrow \text{female}(X), \text{parent}(Y, X)
\]  \hspace{1cm} (2.1)

Positive examples are identified with the symbol \( \oplus \) preceding the predicate and negative examples are identified with symbol \( \ominus \).

In practice, the algorithms developed in ILP focus on empirical inductive logic programming. This was true in 1996 and still applies nowadays, as we are able to verify from the most recent approaches condensed in a recent overview by [Raedt, 2010]. Empirical inductive logic programming systems are characterized by learning single concepts while using batch learners, i.e., requiring all the training ex-

<table>
<thead>
<tr>
<th>Training examples</th>
<th>Background knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \oplus \text{daughter}(Sue, Eve) )</td>
<td>parent(Eve, Sue)</td>
</tr>
<tr>
<td>( \oplus \text{daughter}(Ann, Pat) )</td>
<td>female(Ann)</td>
</tr>
<tr>
<td>( \ominus \text{daughter}(Tom, Ann) )</td>
<td>parent(Pat, Ann)</td>
</tr>
<tr>
<td>( \ominus \text{daughter}(Eve, Ann) )</td>
<td>parent(Tom, Sue)</td>
</tr>
</tbody>
</table>

Table 2.1: Training examples and background knowledge necessary for learning the daughter relation.
amples to be available in order for the learning process to start, and are non-interactive, meaning these systems do not rely on any sort of verification system (usually called an oracle) to verify the validity of generalizations or classifications generated by the learner.

The other research area that is focused on identifying causal effect relations is Pattern Mining. It flourished with the seminal paper by [Agrawal et al., 1993] which introduced the Apriori algorithm for mining frequent itemsets and learning association rules from transactional databases.

The fundamental concepts of pattern mining are the following. Let $I = I_1, I_2, ..., I_m$ be a set of items, i.e., a set of attribute-value pairs. Let $D$ be a set of database transactions where each transaction $T$ is a particular set of items such that $T \subseteq I$. A set of items is referred to as an item-set. An association rule is an implication of the form $A \rightarrow B$, where $A \subseteq I$, $B \subseteq I$, and $A \cap B = \emptyset$. Each rule is characterized by two measures: support and confidence. The first one is related to the expression $sup(A \rightarrow B) = P(A \cap B)$ which corresponds to the percentage of transactions in the database that simultaneously contain both $A$ and $B$. The second measure can be represented by $conf(A \rightarrow B) = P(B \mid A)$ and it is the percentage of transactions containing $A$ that also contain $B$. For example, a rule bread $\rightarrow$ milk [support = 20%, confidence = 80%] means that 20% of all transactions in the database show that bread and milk are bought together and 80% of customers who buy bread also buy milk. In this way, we are able to provide causality connections between items.

Pattern mining has been extensively studied and extended to consider multiple data models (sequences and time series [Agrawal and Srikant, 1995], graphs, tree and lattices [Washio and Motoda, 2003], etc.) which makes it a popular option to deal with the problem of data analysis. Thus, pattern mining algorithms are present in several Business Intelligence tools ([Get, 2013, Cameron, 2009, Witten and Frank, 1999]) making this line of research the most interesting one to be explored.

The major drawback of pattern mining is that it strives for completeness [Zaki, 2012], i.e., discovering all patterns that satisfy certain conditions. But typically many patterns will fulfill the constraints at hand in a given problem. This problem 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.

Some of the strategies that currently exist to mitigate this problem include filtering techniques, whereby the amount of patterns reported are filtered from the original set of patterns found considering some specific criteria (qualitative, cluster-based, top-k patterns). Other approaches to solving the problem of pattern explosion involve compacting the patterns found in an efficient data structure. In the later, the problem is only partially solved since, in this way, only space and time are saved from the usual patterns found. The amount of patterns reported will be the same. Also, in practice, most compaction algorithms incur in lossy compaction strategies to filter out patterns before compaction is performed.

Next, we review two of the most well-known forms of pattern compaction, closed and maximal itemsets.
Table 2.2: Transactional database D for Closed Frequent Itemsets

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C</td>
</tr>
<tr>
<td>2</td>
<td>A B</td>
</tr>
<tr>
<td>3</td>
<td>A B C</td>
</tr>
<tr>
<td>4</td>
<td>B C D</td>
</tr>
<tr>
<td>5</td>
<td>C D</td>
</tr>
</tbody>
</table>

2.1 Closed Frequent Itemsets

The concept of closed frequent itemsets was first introduced independently by [Pasquier et al., 1999] and [Zaki and Ogihara, 1998]. According to the authors, a frequent itemset \( I \) is said to be closed if and only if no superset of \( I \) has the same support than \( I \). It is also important to define the notion of closure, related to the previous definition. The closure of an itemset \( I \), denoted by \( cl(I) \), is the unique maximal superset of \( I \) having the same support as \( I \). A subset \( S \) of \( I \) is then called closed if \( cl(S) = S \).

Representing the set of discovered patterns just by its set of closed patterns is referred to as lossless compression, since for a given support threshold it is enough to know all closed frequent itemsets and their supports in order to infer with total accuracy all frequent itemsets and its corresponding supports. Consider an itemset \( I \). If \( I \) has no superset in the closed set (i.e., the set of all closed frequent itemsets), then it is trivial that \( cl(I) \) is not frequent, and thus \( I \) cannot be frequent either. If on the other hand, \( I \) has at least one superset in the closed set, then the itemset \( X \) that corresponds to the smallest superset of \( I \) in the closed set is the one that verifies \( sup(I) = sup(X) \) [Calders et al., 2006].

As an example consider Figure 2.1 where the itemset lattice of database \( D \) (from Table 2.2) is depicted. The lattice covers all possible combinations of items. Each grey cell corresponds to the frequent itemsets in \( D \) when considering a minimum support of 40% (\( \text{minsup} = 2 \)). On the left side of each cell in the lattice, a sequence of numbers represents the transaction identification numbers (TIDs) in which the itemset of that same cell appears in the database. The lattice is better understood when reading it bottom-up.

When no sequence of TIDs is present in the left of a cell, it means no transaction contains that itemset. In the case of cell \( A \), which represents existence of item \( A \) in a transaction, values 1, 2 and 3 are next to it, which means that item \( A \) is present in transactions with TID equal to 1, 2 and 3. That is in fact the case, as Figure 2.1 shows. In the case of this particular itemset, the color of the cell is grey, meaning it is frequent, which is true since it occurs in two transactions out of five, hence in 40% of the cases (\( sup(ABC) = 2 \geq \text{minsup} \)).

As previously described, closed frequent itemsets are a subset of frequent itemsets, thus the frequent...
itemset analysed before consisting only of item A is a candidate to be a closed frequent itemset. In order for it to be a closed frequent itemset its closure must be itself. That is not the case, since the itemset consisting of A has a superset with the same support (itemset AB). In fact, in this case in particular, if we analyse the lattice, cl(A) = AB since AB is the unique maximal superset of A having the same support as the latter. The same reasoning may be applied to conclude that the closed set of database D consists of all itemsets with thick borders: ABC, AB, BC, CD, B and C.

2.2 Maximal Frequent Itemsets

In an attempt to reduce even further the number of patterns found that are deemed interesting, the notion of maximal frequent itemset was introduced by [Lin and Kedem, 1997]. A maximal frequent itemset is an itemset which is frequent and that has no proper frequent superset. Therefore, it is a more restrictive definition than closed frequent itemset, i.e., all maximal frequent itemsets must be closed frequent itemsets, but the opposite is not necessarily true.

As an example, consider again the itemset lattice of database D in Figure 2.1. Every closed frequent itemset identified is a potential candidate for being classified as a maximal frequent itemset. Itemset CD is a maximal frequent itemset since it has no frequent superset (both its supersets ACD and BCD are not frequent, thus CD is maximal). In the case of itemset AB, something different happens: one of its supersets, ABC, is frequent and therefore AB is not a maximal frequent itemset.

The main reason for creating this new constraint on frequent itemsets was, as previously mentioned, the need of reducing the number of patterns found, but it also came as an excuse to look at the problem of efficiently mining data in a different way, essentially by using pruning techniques that reduced the typical candidate generation that was performed in Apriori-like algorithms, in particular the anti-monotonicity rule, also known as downward closure property, which states that if \( X \subseteq Y \) then \( sup(X) \geq sup(Y) \) (e.g., \( sup(AB) \) is always at least as large as \( sup(ABC) \), since \( AB \subseteq ABC \)).
Chapter 3

Literature Review

In this chapter we review the most relevant work in the stream of pattern compaction. We begin by exploring more elaborate forms of pattern constraints, introduce the concept of top-k patterns and explain the way in which the minimum description length principle was used for reducing the amount of patterns reported to the user. Additionally, we review the way in which clustering techniques [Bishop, 2006] have contributed to solutions in pattern summarization. Following, we explain recent algorithms in Inductive Logic Programming (ILP) that have a resembling goal to ours. We conclude the chapter with a review of three data structures that can be used to store patterns, with two of them compacting the patterns to some extent.

3.1 Filtered Mining

This subsection introduces various approaches related to filtering the patterns found. This is done in such a way that only the most meaningful, useful or interesting patterns are left out. Alternatives within this subsection also involve the creation of patterns capable of representing a given set, considering different criteria.

3.1.1 Constrained Sets

One of the most popular approaches in data mining, and pattern compression in particular, has been to mine only patterns that satisfy certain constraints, in order to reduce the number of patterns that can be seen as possible solutions. These techniques include closed frequent itemsets and maximal frequent itemsets that we already introduced in Chapter 2, but also δ-free itemsets, disjunction itemsets and non-derivable itemsets. In this section we will detail algorithms that belong to the closed and maximal patterns and we also will describe each one of the three last concepts and try to make clear some of its properties, while highlighting the pros and cons of each one.
Algorithms for mining closed and maximal patterns

Pincer-Search [Lin and Kedem, 1997] was the first algorithm to use maximal frequent itemsets mining through the use of a hybrid traversal approach in which the algorithm runs both top-down and bottom-up in the lattice tree. The process of Pincer-Search is based on two crucial observations (derived from the anti-monotonicity rule):

1. If an itemset is infrequent, all its supersets are also infrequent and it needs not be examined further.
2. If an itemset is frequent, all its subsets are also frequent and it needs not be examined further.

The two-way approach taken by this algorithm can speed-up the discovery of the maximal set (collection of all maximal frequent itemsets). It also reduces the number of accesses to the database improving I/O performance and it is faster than previous algorithms especially when the patterns found are long.

MaxEclat and MaxClique [Ogihara et al., 1997] use itemset clustering schemes to generate potential maximal itemsets. MaxEclat clusters itemsets using equivalence classes and applying an hybrid search approach similar to that of Pincer-Search. MaxClique processes maximal hypergraph cliques. The clusters and hypergraph cliques used in the former and latter algorithms are in both cases refined sets of potential maximal itemsets.

Max-Miner [Jr., 1998] attempts to “look ahead” in order to quickly identify long frequent itemsets, by using a heuristic, which can often determine when a new candidate itemset is frequent before accessing the database. This is done based on the information gathered from previous accesses that allow the algorithm to compute a lower-bound on the support of that itemset. Max-Miner distances itself from Pincer-Search in that it uses a simplistic polynomial time algorithm guided by an heuristic instead of an NP hard reduction phase to guarantee no long candidate itemset contains any known infrequent itemset.

Implementations based on bitmap representations of the database have also been successful at mining maximal frequent itemsets while using effective pruning mechanisms to accelerate the process. Both MAFIA [Burdick et al., 2001] and more recently MFIF [Jnanamurthy et al., 2013] are the most well-known examples of that use of an alternative representation approach. In MFIF, the most recent of the works in maximal frequent itemsets, the authors aim at finding the maximal frequent itemsets first by using a novel algorithm to generate combinations of subsets with the most populated transactions in the database.

In SmartMiner [Zou et al., 2002] an augmented heuristic and tail information is used, which has many benefits: it does not require superset checking, it reduces the computation for counting support and yields a small search tree. With all the optimizations and techniques used, SmartMiner incurs in a performance of an order of magnitude faster than MAFIA (Burdick, Calimlim e Gehrke 2001).

δ-Free Sets

A δ-free set [Boulicaut et al., 2000] is defined in terms of the strength of a rule. A rule is said to be strong if it respects some minimum threshold for the support. In this way, a δ-strong rule is one that verifies $X \rightarrow a$, i.e., $sup(X) - sup(X \cup \{a\}) \leq \delta$, where $X \subseteq I$ and $a \in I \setminus X$. What this means is that the support...
of $X$ has to be almost the same as that of generalizing the itemset to also include $a$, i.e., $X \cup \{a\}$ (only
off by a small percentage, $\delta$), thus enabling us to say that "it is almost true that $X \rightarrow a$".

Therefore, the main concept and basic intuition behind the idea of $\delta$-free sets is that if we know that
the association rule $ABC \rightarrow D$ is nearly an exact rule (i.e., it has only a few exceptions), then we can
approximate the frequency of itemset $ABCD$ using the frequency of $ABC$.

MinEx [Boulicaut et al., 2000]) was the first algorithm to use the concept of $\delta$-free sets in order to
generate all free sets of patterns capable of answering frequency queries with a bounded error. It works
in a level-wise fashion starting from singleton itemsets containing only one item each and moving up
the lattice stopping at the level of the largest frequent free sets. At each iteration of the algorithm the
database is scanned to filter out the frequent free sets, generating the candidates for the next iteration,
taking every set of the size of the next iteration such that all proper subsets are frequent free sets.

FTMiner is another algorithm that takes advantage of the concept of $\delta$-free sets, but focuses on
mining large databases. The authors introduce $\gamma$-frequent patterns and use both this definition in combi-
nation with the definition of $\delta$-free to create various pruning criterions mainly based on the fact that both
these measures are anti-monotonous constraints. Let $X$ be an attribute pattern, $O$ an object pattern.
The extension of $X$, denoted $g(X)$, is the maximal set of the objects containing $X$. The key idea is to use
the extension $g(P)$ of a pattern $P$ to check these constraints, because the extension has few objects in
large databases. A $\gamma$-frequent pattern is one that verifies $|g(P)| \geq \gamma$. In their work, the authors point out
that exploration of the use of pattern extensions is a possible way of improving the extraction of patterns
satisfying other constraints [Hébert and Crémilleux, 2005].

**Disjunction-Free Sets**

*Disjunction-free sets*, yet another compact representation of patterns, has been proposed by [Bykowski
and Rigotti, 2001]. It is a generalization of 0-free sets and it is based on the following idea. Let $A$, $B$, $C$
and $D$ represent items in a database. If the rule $A \land B \rightarrow C \lor D$ is true, i.e. if when $A$ and $B$ happen in a
transaction simultaneously, $C$ or $D$ (or both) will also occur in this transaction then the support of $ABCD$
can be derived from the values of the support of $AB$, $ABC$ and $ABD$. This is so, since the sum of the
supports of $ABC$ and $ABD$ is equal to the sum of the supports of $AB$ and $ABCD$. This last itemset is then
called a non-disjunction-free set and representations of patterns are achieved through the use of the
disjunction-free sets. The authors show that using this representation one can regenerate all frequent
itemsets and their exact supports.

The work on disjunction-free sets has been revisited and slightly improved in subsequent works but
the core part of the representation, i.e., the frequent disjunction free sets, remains the same [Calders
et al., 2006].

In practice, the disjunction-free sets used are the ones commonly known as simple disjunction-free
sets of the form $X \rightarrow a \lor b$, where $X \subseteq I$ and $a, b \in I \setminus X$. A generalization of such form, gen-
eralized disjunction-free sets corresponds to a set of rules like $X \rightarrow a_1 \lor ... \lor a_n$ and in this case item-
set $X$ is a generalized disjunction-free set if and only if for any value of $n \geq 2$, there is no valid rule
$X \setminus a_1, ..., a_n \rightarrow a_1, ..., a_n$, where $a_1, ..., a_n \subseteq X$.  

9
### Table 3.1: Transactional database D for Non-Derivable Itemsets.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>sup(A) = 5</th>
<th>sup(BC) = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A C D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A B D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>C D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B C D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B C D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>B C D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A B C D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.2: Supports of itemsets in D.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>sup(B) = 7</th>
<th>sup(BD) = 6</th>
<th>sup(C) = 7</th>
<th>sup(CD) = 6</th>
<th>sup(D) = 9</th>
<th>sup(ABC) = 2</th>
<th>sup(AB) = 3</th>
<th>sup(ABD) = 2</th>
<th>sup(AC) = 3</th>
<th>sup(ACD) = 2</th>
</tr>
</thead>
</table>

Although this method results in a reduction in the number of patterns stored, hence enhancing memory efficiency, it lacks the expressiveness desired for a good understanding of the types of patterns found by the end user. This is mainly due to the fact that if the original amount of patterns is reported to the user, then nothing is gained in terms of pattern representation. On the other hand, if only the disjunction free sets are part of the final sets of patterns found, then it is the user himself who will have to infer the possible patterns that can be generated out of a disjunction-free set.

### Non-Derivable Itemsets

In [Calders and Goethals, 2002] the authors propose a novel concept for condensed pattern representation called non-derivable itemsets. At the time, one of the strongest concerns, that is still a valid one today, was the number of scans needed to perform through the database in order to get all relevant frequent itemsets and their corresponding frequencies, always considering the best minimal representation of patterns found. What the authors aimed to do was to present several new methods to identify redundancies in the set of all frequent itemsets and to exploit these redundancies, resulting in a concise representation of all frequent itemsets and significant performance improvements of the mining operation.

The basic idea behind non-derivable itemsets is that we are capable of establishing lower and upper bounds on the support of the itemsets in a database, ultimately finding the exact value of that support. For example, from $sup(A) = 5$, $sup(AB) = 3$ and $sup(AC) = 4$, we can derive that the support of $ABC$ must be in the interval $[2, 3]$.

As a more complex example, consider the transactional database in Table 3.1. Applying the same reasoning as before, one may construct the mathematical formulas present in Table 3.3 to derive the support of $ABCD$ based on the supports of smaller itemsets (represented in Table 3.2). It is 1, since by rule $R_{AC}$ the lower bound is 1 ($sup(ABCD) ≥ 1$) and by rule $R_A$ the upper bound is also 1 ($sup(ABCD) ≤ 1$).

As you can verify in Table 3.3, the number of rules increases exponentially with the cardinality of the itemset $I \setminus X$. This size $|I \setminus X|$ is called the depth of rule $R_I(X)$. In practice, only rules of limited depth are used to calculate the bounds of the itemsets.

In essence, we call an itemset $I$ a derivable itemset if and only if there is a lower bound $LB(I)$ and an
Table 3.3: Tight bounds on the support of itemset ABCD.

<table>
<thead>
<tr>
<th>Tight bound expression</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sup(ABCD) \geq \sup(ABC) + \sup(ABD) + \sup(ACD) + \sup(BCD) - \sup(AB) - \sup(AC) )</td>
<td>( R_1 )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(A) - \sup(AB) - \sup(AC) - \sup(AD) + \sup(ABC) + \sup(ABD) + \sup(ACD) )</td>
<td>( R_A )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(B) - \sup(AB) - \sup(BC) - \sup(BD) + \sup(ABC) + \sup(ABD) + \sup(BCD) )</td>
<td>( R_B )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(C) - \sup(AC) - \sup(BC) - \sup(CD) + \sup(ABC) + \sup(ACD) + \sup(BCD) )</td>
<td>( R_C )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(D) - \sup(AD) - \sup(BD) - \sup(CD) + \sup(ABD) + \sup(ACD) + \sup(BCD) )</td>
<td>( R_D )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(ABC) + \sup(ABD) - \sup(AB) )</td>
<td>( R_{AB} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(ABC) + \sup(ACD) - \sup(AC) )</td>
<td>( R_{AC} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(ABD) + \sup(ACD) - \sup(AD) )</td>
<td>( R_{AD} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(ABC) + \sup(BCD) - \sup(BC) )</td>
<td>( R_{BC} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(ABD) + \sup(BCD) - \sup(BD) )</td>
<td>( R_{BD} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \geq \sup(ACD) + \sup(BCD) - \sup(CD) )</td>
<td>( R_{CD} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(ABC) )</td>
<td>( R_{ABC} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(ABD) )</td>
<td>( R_{ABD} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(ACD) )</td>
<td>( R_{ACD} )</td>
</tr>
<tr>
<td>( \sup(ABCD) \leq \sup(BCD) )</td>
<td>( R_{BCD} )</td>
</tr>
</tbody>
</table>

upper bound \( UB(I) \) such that \( LB(I) = UB(I) \). Therefore, those itemsets need not be explicitly represented since they can be derived from other itemsets that contribute to both \( LB(I) \) and \( UB(I) \). In this way, it is necessary to have a representation capable of providing such compaction. The NDI-representation is provides that representation and is defined as follows:

\[
NDI(D, \sigma) = \{ (I, \sup(I)) \mid \sup(I) \geq \sigma, LB(I) \neq UB(I) \}
\] (3.1)

From this definition, for every set \( I \) able to be derived, it can be decided whether or not it is frequent. Hence, every itemset that is not in NDI is either infrequent or derivable (or both). In order to know the bounds and ultimately find out what the exact frequencies of the different derivable itemsets are, one needs to know the support of all subsets of \( I \). In an iterative fashion, we calculate the bounds on the subsets of \( I \) and the following situations can occur. If a subset of \( I \) is infrequent, then \( I \) must be too. Otherwise, the supports of all subsets of \( I \) in the border of NDI are known and thus we can calculate the supports of all subsets of \( I \) just above the border, and so on.

### 3.1.2 Top-k Patterns

To deal with the problem of pattern explosion by taking a different approach, whereby one limits the number of patterns in order to find only the \( k \) most important patterns, some algorithms have been proposed since the introduction of such type of pattern mining algorithms.

One of the earliest algorithms of this kind is the one by [Han et al., 2002] whose target top-k patterns are the \( k \) most frequent closed itemsets with a user-specified minimum length \( \text{minlen} \) but without a minimum support \( \text{minsup} \). The authors present TFP, the efficient algorithm for performing such task, which relies on two novel methods to raise the support threshold (closed node count and descendant sum) and to prune the used FP-tree both during and after its construction. Similarly to previously mentioned algorithms, this one uses a combined top-down and bottom-up strategy to mine the FP-tree. Besides
this, an efficient hash-based checking algorithm is employed to verify if a given potential close pattern is in fact closed. Note that the minlen parameter is necessary because without it only subsets of the patterns with that minimum length would be found (since they always have the highest frequency).

Another top-k summarization technique is the one by [Afrati et al., 2004], in which they propose a solution in terms of answering the following question. What are the k sets that best approximate a collection of frequent itemsets? In this algorithm the pattern support is not a summarization criterion. Instead, the high compressibility with maximal coverage is its primary goal. In order to allow for this to happen, the algorithm reports maximal patterns and is permissive (to a certain extent materialized by a threshold $\alpha$) to the presence of some false positives [Hasan, 2009]. As an example, consider that the patterns of a transactional database were $ABC$, $ABD$, $ACD$, $AE$ and $BE$. Then, with the algorithm by Afrati et al., it would be possible to report back as a solution the following patterns: $ABCD$ and $ABE$, which contain two false positives ($BCD$ and $ABCD$). The drawbacks of this algorithm are clear in the paper: it is a post-processing algorithm, meaning it will perform the selection of the top-k patterns only after all patterns (good or bad, i.e., interesting or not) have been found. Also, it is not general in the sense that it will not work as expected for complex patterns, although it is not clear what is considered a complex pattern.

A more recent approach was adopted by [Xin et al., 2006] in which an algorithm to find the redundancy aware top-k patterns is developed. The authors formulate the problem through a general ranking model, which integrates two measures, significance and redundancy, into one objective function. The significance of a pattern represents how useful or interesting it is and redundancy between a pair of patterns is measured in terms of how similar that pair of itemsets is. The algorithm is shown to perform within a logarithmic bound.

In the latest work on this matter [Kameya and Sato, 2013] propose yet another top-k pattern miner, based on FP-growth, called RP-growth which is an efficient algorithm for mining the top-k discriminative patterns which are highly relevant for the class of interest. It conducts a branch-and-bound search using anti-monotonic upper bounds of relevance scores such as F-score and $\chi^2$, and the pruning is translated to minimum support raising (a basic strategy for pruning in top-k mining). The authors also make use of a concept denoted weakness to perform an aggressive pruning strategy enabled by the search tree used in FP-growth. Pattern $p'$ is weaker than $p$ if and only if $p' \supset p$ and $R_c(p') \leq R_c(p)$, with $p, p' \in P$ and $R_c(x)$ being the relevance between a pattern $x$ and the class of interest, $c$. This approach reveals better use of supersets than maximal sets. Supersets are preferred with the sole goal of leading to fewer patterns, while in RP-growth the quality of the supersets is also an important factor (here denoted by relevance). As a limiting factor, this task is better suited for datasets with one (or more) explicit class attributes.

### 3.1.3 Approaches based on Minimum Description Length

A radically different approach to the problem we have proposed to address earlier in this document is proposed by [Siebes et al., 2006] as a way to take the cause of the explosion of patterns head on. In the
work by these authors, they aim at solving the problem with the aid of the minimum description length principle, which follows the Occam’s razor principle. In the context of pattern mining, it means that the best set of frequent itemsets is the one that compresses the database best, i.e., a set of itemsets is interesting if and only if it yields a good (lossless) compression (representation) of the database.

First it is important to define some key concepts in order to understand some details of this algorithm. Given a set of models $H$, the best model $H \in H$ is the one that minimizes $L(H) + L(D|H)$, where $L(H)$ is the length, in bits, of the description of $H$ and $L(D|H)$ is the length, in bits, of the description of the data when encoded with $H$. A set of itemsets $C$ is an itemset cover for a database $D$ if and only if for each transaction $t \in D$ there is a subset $C(t) \subseteq C$ such that:

1. $t = \bigcup c_i \in C(t) c_i$
2. $\forall c_i, c_j \in C(t): c_i \neq c_j \rightarrow c_i \cap c_j = \emptyset$

As an example, consider one would have a database like the one in Table 3.4. Then, we can say the set of itemsets composed only of singleton itemsets, $C = \{A, B, C\}$, is a cover for database $D$.

The authors define a coding scheme (CS) for a database $D$ as a pair $(C, S)$ in which $C$ is an itemset cover of $D$ and $S$ is a function $S : D \rightarrow P(C)$ such that $S(t)$ covers $t$.

A coding/decoding table is a table with three columns, Item Set, Code and Tuple List such that:

1. Itemset and Tuple List form a coding scheme.
2. The length of the Code for $c_i \in C$ is inversely proportional to its frequency (with frequency $freq(c) = \big| \{ t \in D \mid c \in S(t) \} \big|$).

An example of a standard code table is present in Figure 3.1.

The authors then present the Cover algorithm, which in order to code a transaction $t$, takes the first itemset $i$ in the code table that is a subset of $t$ and continues this procedure on $t \setminus i$ in a recursive way, halting when the remainder is empty. One can in this way use Cover and table look-up to code the transactions in a database and simply table look-up to decode a coded transaction.

However, as shown in the paper, this algorithm reveals to be inefficient in a number of aspects and can be improved in several aspects of its functioning. The authors redefine the problem and name it Minimal Coding Set Problem: for a database $D$ over a set of items $I$, find a coding set $C$ for which $L_c(D)$, the length of $D$ when encoded with $C$, is minimal.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C</td>
</tr>
<tr>
<td>2</td>
<td>A B C</td>
</tr>
<tr>
<td>3</td>
<td>A B C</td>
</tr>
<tr>
<td>4</td>
<td>A B C</td>
</tr>
<tr>
<td>5</td>
<td>A B C</td>
</tr>
<tr>
<td>6</td>
<td>A B</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>B</td>
</tr>
</tbody>
</table>

Table 3.4: Transactional database $D$ for Minimum Description Length.
In order to find a solution to this problem, the authors resort to various heuristics that translate into a reasonable relationship between compression size and code table expressiveness: a basic standard coding set construction, an ordering heuristic, a naïve compression based on cover order maximality and an efficient pruning strategy [Vreeken et al., 2011].

### 3.1.4 Clustering-based Approaches

In a way, pattern summarization can be seen as a clustering problem. If one manages to create a good enough distance function between frequent itemsets, reasonable clustering can be achieved with promising results. A compression task can then obtain a set of representative patterns able to describe the most relevant regularities in a database.

#### Greedy Clustering

In [Xin et al., 2005] the authors develop a compression framework for clustering of frequent patterns by pattern similarity, picking a representative pattern for each cluster. As a motivation to the problem, Xin et al. present a transactional database similar to the one in Table 3.5.

The closed itemsets cannot get any compression on this subset. The maximal itemsets will only report the itemset $P_3$. However, we are able to verify that patterns $P_1$, $P_2$ and $P_3$ are significantly different among themselves when considering their support and the maximal set totally loses this information. On the other hand, $P_4$ and $P_5$ are very similar in both expressions and supports. Therefore, the authors consider that a high-quality compression is obtained if for this database, with these patterns and corresponding supports, the output is $P_2$, $P_3$ and $P_4$. 

<table>
<thead>
<tr>
<th>Pattern $P_i$</th>
<th>Items</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>${A, C, D, E}$</td>
<td>205227</td>
</tr>
<tr>
<td>$P_2$</td>
<td>${A, C, D, E, F}$</td>
<td>205211</td>
</tr>
<tr>
<td>$P_3$</td>
<td>${A, B, C, D, E, F}$</td>
<td>101758</td>
</tr>
<tr>
<td>$P_4$</td>
<td>${B, C, D, E, F}$</td>
<td>161563</td>
</tr>
<tr>
<td>$P_5$</td>
<td>${B, C, D, E}$</td>
<td>161576</td>
</tr>
</tbody>
</table>
In order to achieve such result, the authors face various key problems: a valid distance metric is needed to evaluate pattern distance, the quality of the clustering needs to be controllable, the representative pattern should be able to describe both expressions and supports of other patterns and the algorithm has to be efficient. The authors define a valid distance metric which they denote distance between two patterns, similar to Jaccard index, and defined as follows:

\[ D(P_1, P_2) = 1 - \frac{|T(P_1) \cap T(P_2)|}{|T(P_1) \cup T(P_2)|} \]  

(3.2)

In the formula, \( P_1 \) and \( P_2 \) are two patterns in the database and \( T(P) \) is the set of transactions in which pattern \( P \) occurs. This distance metric is proven to be valid such that it verifies a number of necessary constraints [Xin et al., 2005].

With the distance metric defined, the next step is to agree on a clustering criterion. Here, the authors concentrate on the expressions rather than on the support of each pattern. We say \( B \) can be expressed by \( A \) if and only if the items in \( B \) are present in the itemset of \( A \), and it is represented by \( O(B) \subseteq O(A) \), where \( O(P) \) is the set of items in \( P \). Following this definition, assume patterns \( P_1, P_2, ..., P_k \) are in the same cluster. The representative pattern \( P_r \) of the cluster should be able to express all the other patterns. The authors propose using the concept of \( \delta \)-cluster, where \( 0 \leq \delta \leq 1 \) is the tightness measure of a cluster. From the general idea of this concept, we get the following definition: A pattern \( P \) is \( \delta \)-covered by another pattern \( P' \) if \( O(P) \subseteq O(P') \) and \( D(P, P') \leq \delta \). A set of patterns form a \( \delta \)-cluster if there exists a representative pattern \( P_r \) such that for each pattern \( P \) in the set, \( P \) is \( \delta \)-covered by \( P_r \).

With these ground rules defined, the authors continue to present an algorithm RPglobal which assumes all the frequent patterns are pre-computed (it is a post-processing algorithm), it needs to find the globally best representative pattern at each step and needs to compute the pair-wise distance between all frequent patterns. All these factors lead to an expensive cost w.r.t. to complexity of the algorithm and time of processing. This lead the authors to develop an alternative algorithm RPlocal which sacrifices the theoretical bounds on the number of output or distance among any pair of patterns in a cluster with the goal of enhancing the efficiency of the search.

RPlocal finds a locally good representative pattern at each step directly mining from raw data and not computing the distance between patterns in a pair-wise fashion. Hence, RPlocal avoids huge intermediate outputs (happens in post-processing algorithms) and allows for more efficient pruning methods. The authors go even further proposing an hybrid approach in which RPglobal and RPlocal are combined to form RPcombine, which acts as a balancer enabling the user to tune the trade-off between efficiency and performance the way it best suits him. RPcombine first uses RPlocal to get a small subset of candidate representative patterns and then uses RPglobal to find the final results.

More recently, [Liu et al., 2012] introduce MinRPset, an algorithm similar to RPglobal but using several techniques to reduce running time and memory usage. It makes use of a data structure for storing frequent patterns, called CFP-tree, which we will explain in detail in a later section of this document. This tree structure also supports efficient retrieval of patterns that are \( \delta \)-covered by a given pattern. In the same paper, the authors present an alternative algorithm, FlexRPset, which is developed based on
MinRPset but works with an extra parameter $k$ (to control the minimum number of times that a frequent pattern needs to be covered) allowing users to make a trade-off between efficiency and the number of patterns selected. With the decrease of $k$, FlexRPset becomes faster, but it also produces more representative patterns reaching its best performance (surpassing RPlocal w.r.t. running time) when $k=1$. It still produces fewer representative patterns than RPlocal in almost all cases. In that work the gain is not so much in terms of innovation but more in terms of usefulness of the algorithm: with the introduction of the CFP-tree users are able to retrieve the support of the patterns from that data structure directly.

Profile-based Clustering

After exploring the application of clustering methods to pattern mining and in particular its relationship with pattern compression in the aforementioned RPglobal and RPlocal algorithms, [Yan et al., 2005] proposed profile-based pattern summarization as an attempt to effectively integrate a frequency criterion with the coverage criterion, i.e., taking into account both the expression and the support measure (or frequency).

The authors’ representation of patterns is derived from the $k$ clusters formed by all frequent itemsets, which are covered and estimated by a pattern profile that comprises three components. The first is an itemset consisting of the union of all the frequent itemsets in the cluster and is called master pattern, e.g., if a cluster contains the itemsets $AC$, $ADE$ and $CFG$ then the master pattern of this cluster is $ACDEFG$. The second component is the total number of transactions in which the itemsets of that cluster occur, denoted $T$. The third component is called distribution vector and it is a recording of all relative frequencies for each individual item in the set of transactions in which the itemsets of the cluster occur.

Given these rules, the way one estimates the frequency of itemset $AC$ is by computing the following: $p(ACDEFG) \times p(A) \times p(C)$, where $p(ACDEFG)$ is the support of $T$ in the database and $p(A)$, $p(C)$ are the relative frequencies for items $A$ and $C$ in $T$. It is then possible to recover frequent patterns as well as their supports.

Partitioning is done in $k$ groups such that the similarity within each group is maximized and the similarity between the groups is minimized. The authors apply a hierarchical agglomerative clustering method, in which a dendrogram is produced resulting from the merge of two clusters at each level. This structure enables the user to perform pattern searches in a top-down fashion and provides a global view of the patterns. The authors apply three different techniques that contribute to a better performance of the overall algorithm:

1. A major computation cost is the pair-wise calculation of the KL-divergence\footnote{Kullback-Leibler divergence between the distribution vectors in the profiles of a pair of clusters is used to calculate the distance between them, hence allowing the algorithm to decide whether to merge the pair or not.} used to merge clusters, which can be reduced by adopting a $k$-means clustering procedure, thus enabling very fast clustering for a large number of patterns.

2. Based on a lemma that comes as a consequence of using KL-divergence, the authors are able to state that clustering based on frequent itemsets is the same as the clustering based on closed fre-
quent itemsets. Therefore, they summarize closed frequent itemsets instead of frequent itemsets, reducing the number of patterns involved in clustering.

3. Profile updating dominates the computation of the algorithm. A potential solution is to mix two profiles without checking the original dataset or weigh each pattern’s profile equally. This approximation comes as a trade-off between efficiency and summarization quality.

The authors also show that it is possible to determine the optimal value of $k$ by monitoring the change of the support restoration error (estimation error). The greatest disadvantage of this algorithm is that it is a post-processing one, disallowing the mining process to occur in parallel with profiling.

3.1.5 ILP-based Approaches

We now describe the work developed in Inductive Logic Programming. Although our work does not attempt to follow the ideas of this area of interest, its work cannot be left unmentioned due to the similarity of goals pursued. The use of variables as a way of abstracting concrete concepts in more general ones serves as inspiration to the final solution presented in this document.

As [Raedt, 2010] stated in his recent overview, “one of the simplest and best-known inductive logic programming systems is FOIL”. This ILP system follows a basic specialization technique called top-down search of refinement graphs, in which the top-down learners start from the most general clause and iteratively refine the hypothesis until no negative examples are covered. For example, consider that the hypothesis $H$ starts with the clause $\text{daughter}(X, Y) \leftarrow$. In this case, two negative examples are covered (consider once again Table 2.1). In the next iteration, the refinements considered are $\text{daughter}(X, Y) \leftarrow \text{female}(X)$ and $\text{daughter}(X, Y) \leftarrow \text{parent}(Y, X)$. Both these clauses cover one negative example each, but they have a common refinement that becomes the solution to this problem, since it covers no negative examples:

$$H = \{ \text{daughter}(X, Y) \leftarrow \text{female}(X), \text{parent}(X, Y) \} \quad (3.3)$$

In FOIL, the search of refinement lattices is done recurring to heuristics. The key differences between FOIL and its propositional predecessors are the representation and the operators used to compute the specializations of a clause, allowing one to specialize clauses using substitution or by adding an atom to the body (conditional part) of the rule.

Another algorithm and overall framework called Warmr was proposed by [Dehaspe and Toivonen, 1999]. It is a general purpose inductive logic programming system that addresses what the authors call frequent query discovery, which is “a very general Datalog formulation of the frequent pattern discovery problem.” The authors argue that Warmr offers enough flexibility to be tested in both standard and novel settings not supported by special purpose algorithms. It also provides a representation that enables the user to better understand the frequent pattern discovery domain.

Datalog is a subset of clausal logic (and Prolog) that is restricted to function-free definite clauses. A deductive Datalog database is a set of definite clauses. The data mining task of discovering frequent
patterns in Datalog is then the task of discovering frequent queries.

Warmr can be seen as "the inductive logic programming upgrade of Apriori" [Raedt, 2010] in the sense that it also comprises a phase of candidate generation performed in a similar fashion as that of the Apriori algorithm. It starts with a set of candidate queries and evaluates each one of them w.r.t. frequency, classifying them in frequent or infrequent queries, generating new candidates by adding one atom to the query at a time (specializing it in terms of covered knowledge). In the end, the set of frequent queries is returned.

In [Raedt and Ramon, 2004], the authors present an extension to Warmr, called c-armr. The idea behind c-armr is that most inductive logic programming systems overlook background knowledge during the task of hypothesis generation. Because of the absence of background knowledge during this process, many clauses will often be irrelevant since they will be semantically equivalent to other clauses previously considered, thus generating redundant hypothesis whose only contribution is generating unnecessary overhead. The authors introduce the notions of semantic closeness and semantic freeness as a way to circumvent this problem. In this work, the authors also present the concept of \( \delta \)-free clauses, whose idea is that a (small) number of exceptions are allowed in each association rule (similar to \( \delta \)-free sets presented before but now applied to the domain of inductive logic programming). In essence, c-armr is a constraint-based frequent query mining system.

Following the previously described work, Classic'cl was introduced [Stolle et al., 2005], an inductive logic programming system that incorporates various ideas from previous systems and different algorithms such as Claudien, Warmr, c-armr, CLLPAD. A wide range of constraints is integrated, including minimum frequency, maximum generality, exclusive disjunctions and condensed representations.

### 3.2 Pattern Storage

This document has been focusing essentially in techniques to reduce the number of patterns found in a database, but we have not looked at another form of compression in detail: the compression of data itself w.r.t. its internal representation. We have mentioned CFP-trees before but we have not explored the concept and a good enough explanation of what this structure is has not yet been given. In this section, we present the data structures that are most well-known and interesting, while having performance and efficiency as their main concerns. We also explain each one of them taking into account particular features and advantages as well as their drawbacks.

#### 3.2.1 FP-trees

The FP-tree, short for Frequent Pattern Tree, is an extended prefix tree structure storing relevant, quantitative information about frequent patterns [Han et al., 2000]. It is used as part of the process of mining frequent patterns during the FP-Growth algorithm.

The design of this data structured is based on the following crucial observation. If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be
merged using one prefix structure. If we sort the frequent items by their descending frequency, we increase the chances of prefix strings being shared.

Each entry in the frequent item header table consists of an item name and a link to the first node in the tree representing the corresponding item.

We may construct a Frequent Pattern Tree using the following steps. First, we scan the database in order to infer the list of frequent items. In the example from Table 3.6, we would get \((f:4, c:4, a:3, b:3, m:3, p:3)\) for \(\text{minsup} = 3\). Next, we must scan the database a second time. The scan of the first transaction leads to the creation of the first branch of the tree, for the path corresponding to that itemset, \((f:1, c:1, a:1, m:1, p:1)\). The second transaction, \(fca\), shares a common prefix, \(fca\), with the existing path \(fcamp\). The branch then forks at \(a\) and the counters for each node are updated. After processing the second transaction we get \((f:2, c:2, a:2)\) and two subtrees forking at \(a\), each one with \((m:1, p:1)\) and \((b:1, m:1)\), respectively. Figure 3.2 shows the final result.

The FP-tree consists of two main components: a set of item prefix subtrees as the children of the root node and a frequent item header table.

Each node in the set of item prefix subtrees consists of an item name, a counter and a link. The item name is used to identify which item this node is representing. The counter registers the number of transactions represented by the path reaching this node. The link points to the next node in the tree representing the same item.

Benefits from using such a data structure include the need to only scan the database twice, the fact that it is a highly compact structure and containing enough information for the task of pattern mining. However, the patterns found comprise redundancies that are naturally overlooked during the process of construction and subsequent mining.

### 3.2.2 CFP-trees

CFP-tree (Condensed Frequent Pattern Tree) is one of the most promising data structures used for pattern representation in an efficient manner both in terms of mining the database for patterns above a certain threshold or patterns that respect a particular constraint. It is a compact data structure that allows storing and querying of frequent itemsets on disk. CFP-trees were introduced by [Liu et al., 2003] and with it some algorithms were presented proposing well-defined rules to construct the tree and query it both in terms of support (a minimum user-specified threshold) and expression (all patterns that contain...
An example of a database and corresponding CFP-tree is depicted in Figure 3.3. Consider the CFP-tree represented on the right. Each node in that tree is represented as a variable length array where each item in that node is stored in ascending order of frequency. A path from the root to a specific node, e.g., from the root to node 6, represents a pattern (or more than one, if it is not a singleton node). In the case of node 6 we have the pattern $FM$ and the pattern $FA$. Notice that $FM$ is further refined to $FMA$, but by the anti-monotonicity rule if $FMA$ is frequent $FM$ must be frequent as well. It is easy to draw from the example that all entries in the same node share the same prefix.

In a CFP-tree, any entry $E$ in a node comprises four elements:

1. The last element of the path from the root to that entry.
2. The support of the pattern formed by that path.
3. A pointer to the sub CFP-tree originating from that entry (children).
4. A hash bitmap (not represented in Figure 3.3).

As a method for improving efficiency, the authors introduce a hash bitmap in each entry $E$. For every item $i$ in the subtree pointed by $E$, the $j\text{th}$ bit of the bitmap is set to 1, where $j = i \mod N$ and $N$ is the length of the bitmap. In this way, when a search is to be performed, one can first check the bitmap to see whether all items being search are in the sub CFP-tree.

In order to construct the tree, find patterns that respect a minimum support or contain specific items, the authors define a set of steps. We begin by explaining the algorithm for querying the CFP-tree for patterns with a minimum support. Then, we show how to search for patterns with specific items and after that we will show how the tree construction is performed.

The desired output of a query with minimum support constraint is all frequent itemsets that respect a user-defined minimum threshold $\minsup$, which cannot be less than the constructing support of the CFP-tree. According to what has been stated before, the ascending order of supports of the entries in each node allow us to access the first entry that respects that $\minsup$ and, without having to do an explicit check, access all entries in that node that follow the first matching entry. It is a simple algorithm because of the topology of the tree.

The algorithm developed to query patterns with item constraints has a lot to do with the use of the hash bitmap in each entry. In order to reduce unnecessary tree traversals, one can check the bitmap for the presence of the items in the constraint and decide to continue the search or prune a subtree. As previously explained, the bitmap works as a bloom filter [Bloom, 1970] which means that false positives
are possible but false negatives are not, i.e., its use may still lead to unfruitful subtree searches but never to misleading results.

The CFP-tree construction is performed with only two database scans through the use of a pattern growth approach. In the first database scan, all frequent items are mined and sorted in ascending order w.r.t. their frequencies. In the second database scan, a conditional database is created for each item belonging to the set of all frequent items, say $F$. All infrequent items in each transaction are discarded and the remaining items are sorted according to their orders in $F$. If the first item of a transaction is $i$, then that transaction is placed into the conditional database of item $i$. Additional mining can be performed over the resulting conditional databases.

In essence, the CFP-tree is a data structure for efficient compaction of patterns in transactional databases that allows for a condensed representation of frequent itemsets, thus improving memory utilization as well as performance in what concerns querying patterns taking into account minimum support or the presence of specific items. It is very similar to an FP-tree used in the FP-Growth algorithm, nonetheless.

### 3.2.3 Suffix Trees

A suffix tree is a tree-like data structure representing all suffixes of a string. An example of a simple suffix tree for the string CAGAAGT can be seen in Figure 3.4 ($\$\$ denotes the termination of the string).

Ukkonen introduced a linear-time online algorithm for mining the substrings of a string using suffix trees in [Ukkonen, 1995]. Substring mining can be seen as a type of sequence mining, i.e., a mining task where relevant patterns appear in a sequence. Ukkonen’s algorithm was important because it combined various techniques that allowed it to reduce its running time from $O(n^2)$ or $O(n^3)$, in the most naïve implementation, to linear time, i.e., $O(n)$, with $n$ being the length of the string.

In essence, the use of suffix trees, alongside with Ukkonen’s algorithm, makes up for a valid and
efficient option for representing and mining substrings of a string.

In practice, however, and taking into account the problem we aim to solve, this type of data structure reveals to involve unnecessary effort. In this work, we will try to compact transactional patterns (i.e., no repetitions of items are allowed within the same pattern), which means that we will not be focusing in sequences (in which case, repetitions would be allowed) and therefore it will not be the goal of our work to compact sequences of patterns. For example, we will never have to compact patterns like $ABCABAD$, because items $A$, $B$, $C$ and $D$ we will have in the “worst-case scenario” a pattern made up of all the items appearing once and only one: $ABCD$. Also, the anti-monotonicity rule makes it obvious that if we store $ABC$ in the data structure, then $BC$ should not be stored, since the presence of $ABC$ would by itself indicate that $BC$ also respects the minimum support threshold.

These reasons, combined with the fact that this data structure is not ready or thought of to store the kind of patterns we will generate in our solution, makes us conclude that its use would not be adequate for the problem at hand.
Chapter 4

Meta-Patterns and RECAP Algorithm

In this chapter, we present our approach to address the problem of pattern explosion. First, we clearly state the problem we are trying to mitigate with this work. Furthermore, we present a set of requirements we consider essential for the system to be both useful and distinguishable from previous works, while providing the user with different reasonable options when post-processing the patterns found. Following, we present the final architecture as well as the main steps behind our solution. We then demonstrate the flexibility of our algorithm to deal with different types of patterns. Finally, we conclude this chapter with the explanation of optimizations used to enhance the performance of our solution.

4.1 Problem Statement

Consider a database and one of the commonly used pattern mining algorithms (Apriori [Agrawal and Srikant, 1994] or FP-Growth [Han et al., 2000], 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 4.1. 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 4.1. 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: \textit{constant meta-items}, \textit{disjoint meta-items} and \textit{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 4.1, 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 4.1 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., Chips = 2, Sodas = \{4, 5\}).

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

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.

4.1.1 Requirements

In this section we introduce a set of requirements that must be met in order to have a correct, efficient and useful system for the end-user. The presented requirements come in the form of constraints, i.e., rules that must be followed and taken into account when developing our final solution. These will serve as guidelines to features for the system, which will have a direct relation with the concepts presented.

From the description of the formulated problem, we can understand that a meta-pattern is able to represent a set of patterns that share some regularity. However, it should also be clear that there are sets of patterns that are not able to be represented by a unique meta-pattern.

Indeed, in order to be able to compact the entire set of patterns discovered on a given dataset, the patterns reported by the mining algorithm have to be reorganized in such a way that patterns are grouped together considering regularities, in particular among shared attributes. Otherwise, if the patterns were randomly distributed, we would have a combinatorial explosion in the number of possibilities to distribute patterns in sets that share the same characteristics (whichever they are; constant items between those patterns, disjoint relations between items in that group of patterns, or simple linear regression models). It is then important to reorganize the patterns in such a way that the task of finding generalizations between the sets of patterns is eased. For example, a generalization between Diapers = 1, Beers = 2 and Chips = 2, Sodas = 4 would not be successful simply due to the fact that these patterns do not share any attributes.

Requirement 1. Patterns must be organized in a way that reduces the amount of effort needed to perform generalizations. Patterns should be grouped so that the search of regularities is limited to each group in which it makes sense to find those regularities.

Another thing we note when discussing regression meta-patterns, is that it is clear that the definition at hand only makes sense if we are in the presence of numeric data. In particular, two numeric attributes must be present in order for this type of meta-patterns to be found (we must have at least one dependent and one target attribute, both having to be numeric). This leads to our next requirement.

Requirement 2. Simple linear regression requires the existence of at least two numeric attributes.

It can happen the case in which, even though the attributes map to numeric data, the user knows it is
not beneficial (given the specific domain of the problem at hand) to try to find regularities or expressions that describe any of those attributes in terms of another attribute, in that set of patterns.

**Requirement 3.** We must have a way to inform the system about which attributes shall be used when considering generalization using simple linear regression.

If we analyse the process of meta-pattern generation, in particular the simple linear regression part of it, we can think of generalization in two ways. We might be interested in inferring expressions that give the user the capability of thinking more broadly about the patterns found, allowing some error in terms of the expressions found, but inherently generating more abstract and compact descriptions. Or we can be so restrictive that the process of compaction will suffer in terms of expressive power (thus not presenting the user with relations that exist but would not necessarily contribute to a perfect lossless compaction strategy).

Are we willing to lose the expressiveness of the lossy meta-pattern that could be generated, in detriment of creating no generalizations at all? This is a question we believe only the user is capable of answering, depending on the domain at hand and on the goal of that particular user.

As you can see from Table 4.2, we would be able to generate a compact version of the patterns shown, if we were to allow a slight error on regression (because of pattern \textit{Diapers} = 3, \textit{Beers} = 5, \textit{Chips} = 4, in which the relationship \textit{Beers} = 2\textit{Diapers} only fails by a small offset — if attribute \textit{Beers} took value 6 instead of 5 then no regression error would be present). Also, another type of error introduced has to do with the representation itself. It is true that the values of \textit{Chips} only vary between 3, 4 and 7. However, this is only true if we consider the patterns as a whole, i.e., if we were to generate all possible patterns from the meta-pattern that we identify, we would get the patterns in the "Simple unfolding" part of Table 4.2. This would be deceitful for the end-user if he is not familiar with the system and may come as a disadvantage in understanding the patterns found. But it is also an arguably interesting way of understanding relationships between patterns that would otherwise be left unseen.

<table>
<thead>
<tr>
<th>Original patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Diapers} = 1, \textit{Beers} = 2, \textit{Chips} = 3</td>
</tr>
<tr>
<td>\textit{Diapers} = 2, \textit{Beers} = 4, \textit{Chips} = 7</td>
</tr>
<tr>
<td>\textit{Diapers} = 3, \textit{Beers} = 5, \textit{Chips} = 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lossy representation with regression error</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Diapers} = {1, 2, 3}, \textit{Beers} = 2\textit{Diapers}, \textit{Chips} = {3, 4, 7}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple unfolding</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Diapers} = 1, \textit{Beers} = 2, \textit{Chips} = 3</td>
</tr>
<tr>
<td>\textit{Diapers} = 2, \textit{Beers} = 4, \textit{Chips} = 3</td>
</tr>
<tr>
<td>\textit{Diapers} = 3, \textit{Beers} = 6, \textit{Chips} = 3</td>
</tr>
<tr>
<td>\textit{Diapers} = 1, \textit{Beers} = 2, \textit{Chips} = 4</td>
</tr>
<tr>
<td>\textit{Diapers} = 2, \textit{Beers} = 4, \textit{Chips} = 4</td>
</tr>
<tr>
<td>\textit{Diapers} = 3, \textit{Beers} = 6, \textit{Chips} = 4</td>
</tr>
<tr>
<td>\textit{Diapers} = 1, \textit{Beers} = 2, \textit{Chips} = 7</td>
</tr>
<tr>
<td>\textit{Diapers} = 2, \textit{Beers} = 4, \textit{Chips} = 7</td>
</tr>
<tr>
<td>\textit{Diapers} = 3, \textit{Beers} = 6, \textit{Chips} = 7</td>
</tr>
</tbody>
</table>

Table 4.2: Meta-pattern generation for lossy representation and regression error.
**Requirement 4.** The user shall be able to decide on the extent of the regression error (if any) used when generating regression meta-patterns. The error he is willing to allow in the computation shall be given as an offset between the true value and the computed value taking into account the regression parameters found (i.e., slope and intercept).

**Requirement 5.** The user shall be able to choose between a lossless representation on regression meta-patterns and a lossy representation.

The trade-off between expressiveness and accuracy will be further discussed in the Case Study, where our approach is validated.

The other existing systems or algorithms to deal with the problem of pattern explosion (mainly profile-based approaches or clustering solutions that try to create a generalized representation of a set of patterns) are flawed in what concerns the ability (in this case, absence of it) to recreate the patterns found (what we call **unfolding**), i.e. generating the corresponding patterns from a general expression. In our system, we propose to empower the user with this capability, and thus define it as another requirement.

**Requirement 6.** A user shall be able to **unfold** a meta-pattern, i.e., to generate (or recover) the patterns represented by a meta-pattern.

We can now formulate our problem. Given a set of patterns (and necessary compaction parameters), find a set of meta-patterns that is able to describe the original patterns in a more compact way without losing considerable amounts of information.

Considering these requirements and the overall proposal presented before, next we explain the algorithm that makes it possible.

### 4.2 RECAP Algorithm

The problem we aim at solving is that of automatic inference of sets of meta-patterns that describe a given set of patterns. The RECAP algorithm (Regression CompAct Patterns) we are proposing acts in two phases. The first phase involves solving what we previously identified as Requirement 1. 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).

The second phase, the meta-pattern generation step of the algorithm, will act over the groups of patterns previously identified in order to infer the relationships between attributes in the patterns of those groups, and it consists of two procedures.

Figure 4.1 illustrates the overall process of our algorithm. It receives a set of patterns that will be post-processed in what is called the grouping phase. The patterns are grouped considering a particular criteria defined in the **Grouper** for that data model (e.g., for transactional datasets the frequent itemsets are grouped considering a particular criteria, whereas sequential patterns might be grouped in a different way). For each one of those groups, we then enter the generation phase of our algorithm. At this stage, a specific **Generator** will try to generate meta-items with a simple linear regression model. For
the attributes in which simple linear regression is not successful, i.e., if we cannot generate a simple linear regression model for any attribute in the group of patterns, we will try to generate disjoint relations between those attributes. If that is not successful either, we end up with meta-patterns that are replications of the simple patterns themselves. In any case, the final solution for each group will consist of one or more meta-patterns, which will then be grouped together to form the final answer.

4.2.1 Grouping phase

We now explain the logic behind the grouping phase when applied to the specific case of transactional datasets. We then transpose that rationale to explain the differences and similarities that relate this data model to the sequential one.

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 4.1. 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 between them, since it won't produce any results.

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 and we define a distance measure as the criteria to decide whether a pattern should be included in a given group of patterns. This distance measure is computed comparing the support of the first pattern 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
Algorithm 1 Transactional Grouper — grouping phase algorithm pseudo-code for grouping transactional patterns

Require: patterns, \( \Delta \sigma \)
Ensure: list of groups of patterns grouped by attributes and support

\begin{algorithmic}
\State for pattern in patterns do
\State uniqKey \leftarrow generateUniqKey(pattern)
\State groups \leftarrow groupedPatterns.get(uniqKey)
\State group \leftarrow findClosest(pattern, groups)
\If{distance(pattern, group) > \( \Delta \sigma \)}
\State group \leftarrow newGroup(pattern)
\EndIf
\State group.add(pattern)
\EndFor
\State return listGroups(groupedPatterns)
\end{algorithmic}

That group.

The formula for this measure is given by equation 4.1:

\[ d = \left| \frac{\text{reference} - \text{support}}{\text{reference}} \right| \] (4.1)

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, i.e., the Transactional Grouper, the behaviour will be that of the pseudo-code in Algorithm 1.

In our particular implementation of the Transactional Grouper, we propose to store the groups of patterns in an hash table, generating a unique key for each pattern (line 2), which consists of sorting and concatenating the names of the attributes of that pattern. The sorting is necessary because the patterns returned by frequent itemset mining algorithms usually give ids to the items as they appear in the dataset, ignoring the attributes shared, so it is possible that for example Diapers = 1 and Diapers = 2 have completely different ids and appear at very distinct positions in the patterns discovered. By using the unique key, patterns Diapers = 1, Beers = 2 and Beers = 4, Diapers = 2 will generate the same unique key BeersDiapers, as desirable.

After this, we find the corresponding group of patterns for that key in a hash table, with each key corresponding to the previously explained unique key of attributes and value corresponding to a list of patterns that share the same set of attributes (line 3). An example of the hash table used and its contents is depicted in Figure 4.2.

Each one of those groups will be close or far apart from the newly analysed pattern in terms of support value. We will then place it in the group that it is closer to (line 4), or generate a new group in case it is not close to any of the existing ones (line 6). We then add the pattern to the group it belongs to (line 8).

At the end, a list containing each group of patterns is returned to be processed by the next step of
our algorithm (line 10).

What is important to note at this stage is the generality and flexibility of the design of the algorithm. The user can create its own Grouper, taking into account particular nuances of the problem at hand. Although we provide this implementation as a starting point, different users might be interested in grouping patterns in different ways. For example, people might not be concerned with the proximity of patterns in terms of support, which might lead them to disregard the existence of different groups for the same set of attributes (we allow that option in our implementation, by simply making $\Delta \sigma$ virtually infinite, e.g., by setting it at 999%).

### 4.2.2 Generation phase

After the patterns are grouped together, we can perform generalizations on each one of those groups in order to infer the corresponding meta-patterns. These generalizations will take into account the previously defined requirements which will influence the final outcome of this phase of the algorithm and thus, the final answer presented to the end-user.

First, we perform an analysis on the group size and on the number of attributes it comprises. 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 that can assume multiple values for a single attribute, which leads us to create a single disjoint meta-pattern (simply consisting of a disjoint meta-item on that attribute). These two cases are depicted in Tables 4.3 and 4.4, respectively.

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. First, we try to find relations among attributes considering what we call a Simple Linear Regression Model. If the use of this model is unsuccessful, we apply only the Disjoint Model, which is able to group patterns

<table>
<thead>
<tr>
<th>Original pattern</th>
<th>Disjoint meta-pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diapers = 1, Beers = 2, Chips = 3</td>
<td>Diapers = 1, Diapers = 2, Diapers = 3</td>
</tr>
</tbody>
</table>

Table 4.3: Constant meta-pattern generated for single pattern.

<table>
<thead>
<tr>
<th>Original patterns</th>
<th>Disjoint meta-pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diapers = 1</td>
<td>Diapers = {1, 2, 3}</td>
</tr>
<tr>
<td>Diapers = 2</td>
<td></td>
</tr>
<tr>
<td>Diapers = 3</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Disjoint meta-pattern generated for group of 1-itemset patterns.
that only differ in one attribute in a meta-pattern thus consisting only of constant meta-items and one disjoint meta-item. If neither of the models is applicable, we finish the process with the same amount of patterns (i.e., constant meta-patterns) that we began with.

**Simple Linear Regression Model**

The simple linear regression model consists in trying to describe attributes in terms of other ones (the best ones, according to the simple linear regression used).

This causes us to have to define a dependent variable. If a useful attribute is found, i.e., if we are able to describe the dependent variable 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 (explanatory) 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 procedure 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 \(i\) 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 4.5). 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 4.5, 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 4.5 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.

Continuing with the pseudo-code, after performing the regression computations, we decide if a con-

<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, Lettuces = 0.66Beers</td>
<td>Diapers = {1, 2}, Beers = 2Diapers, Lettuces = 3Diapers</td>
</tr>
<tr>
<td>Diapers = 2, Beers = 4, Lettuces = 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Indirect dependency example.
**Algorithm 2** Simple Linear Regression Model — Regression meta-patterns algorithm pseudo-code

**Require:** \( \text{patternGroup}, \varepsilon \)

**Ensure:** \( \text{metaPattern} \)

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

stant meta-item or a regression meta-item shall be created (lines 5 through 11). If regression is not successful, then \( i \) is deemed independent (line 13).

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 \( \varepsilon \)), according to Requirement 4.

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*.

Consider the example in Table 4.6. The algorithm starts by choosing the first attribute to be the dependent one, which means we will try to describe the first attribute in terms of a second one. This will be repeated for every attribute in the group of patterns. This generalization can only be carried out if the "candidate dependent attribute" at hand (i.e., the one we are trying to generalize) is not already
classified as independent or target (initially no attributes are independent, dependent or target).

In this particular example, we would choose attribute Diapers and perform a regression run taking into account as dependents an empty set, since there are no previously defined dependent attributes. We can see that in this case, regression would be successful on attributes Beers and Chips, with the algorithm choosing Beers simply because of the order in which they are tested. The algorithm is then successful (line 4), which will lead to the analysis of the slope. In this case, the slope is not zero (it is in fact 2, since Beers takes a value which is twice as much as the value of Diapers). We then set attribute Diapers as dependent (line 8) and attribute Beers as target (line 9), generating the corresponding regression meta-item (line 10). In the next iteration of the cycle, we would try to generalize the second attribute, Beers, but it is already defined as a target attribute, so we do not try to generalize it. We move on to the third attribute, Chips. We try to generalize it taking into account as dependents the set of attributes containing only Diapers. This means that regression is not going to try to describe Chips in terms of Diapers, since we know that there is already an attribute capable of describing it (thus avoiding what we previously identified as "indirectly dependent generalizations"). Regression is then successful with slope different from zero (being 3, since Chips takes a value which is three times the value of Diapers). Dependent and target variables are updated and a new regression meta-item is created. We then try to generalize the fourth and final attribute. Sodas, however, cannot be described in terms of any other attribute, which will lead it to be set as independent (line 13). At the end, we do not have all attributes set as independent and thus we are able to generate disjoint meta-items for each independent and target attribute (lines 20 through 24), leading to the meta-pattern in Table 4.6.

Note that this explanation is given considering the lossy compaction strategy that we discussed in Requirement 5. If a lossless compaction strategy was to be used, then we would have to check if the total amount of patterns that result from the unfolding process correspond to the original amount of patterns. If that would be the case, then the process of meta-pattern generation would be successful. Otherwise, it would fail.

**Disjoint Model**

We now explain the strategy followed when simple linear regression run is not successful in defining any attribute in terms of another one. As an example consider the case of Table 4.7. The set of patterns in that group is not generalizable considering simple linear regression. However, the patterns can still be generalized considering a disjoint relation in one attribute. In fact, two groups can be further inferred if we split the group in regards to the "more decisive" attribute. The idea followed is similar to (the inverse of) information gain in decision trees, but here it is simplified by using only the relative frequency of items.
Algorithm 3 Disjoint Model — Disjoint meta-patterns algorithm pseudo-code

Name: disjointMetaPatterns
Require: groups, bitmap
Ensure: modifies list of metaPatterns
1: if allSet(bitmap) then
2:   metaPatterns.add(result(groups))
3: else
4:   for group in groups do
5:     best = bestAttribute(group, bitmap)
6:     bitmap.set(best)
7:   for pattern in group do
8:     val ← pattern.attribute(best)
9:     distincts.get(val).add(pattern)
10:   end for
11:   newGroups ← distincts.vals
12:   disjointMetaPatterns(newGroups, bitmap)
13: end for
14: end if

Name: result
Require: groups
Ensure: returns a list with a metaPattern for each group
15: for group in groups do
16:   if group.size = 1 then
17:     metaPatterns.add(constantMetaPattern(group))
18:   else
19:     metaPatterns.add(disjointMetaPattern(group))
20:   end if
21: end for
22: return metaPatterns

in the group of patterns.

In the case of Table 4.7, we should separate the patterns in two finer-grained groups, considering the Diapers attribute. If one was to choose the attribute Beers, we would generate three different groups. But by choosing Diapers, we get two groups resulting in two different meta-patterns.

The algorithm to perform the task of applying a Disjoint Model to the patterns in each group is presented in Algorithm 3. It is a recursive algorithm that will eventually lead to the generation of a set of disjoint and constant meta-patterns. The algorithm is given a list of groups of patterns and a bitmap.

We start out with a list of only one element, which is the group of patterns for which the simple linear regression model was not successful; and a bitmap with each bit corresponding to an attribute index. The bitmap will indicate, at each step of the algorithm (i.e., at each recursive call), which attributes have already been used to split a group in two or more groups.

First, we check if all but one attribute bits are set (line 1). If this is true (line 2), then we are ready to

<table>
<thead>
<tr>
<th>Patterns</th>
<th>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 = 5</td>
<td>Diapers = 2, Beers = 5</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>

Table 4.7: Group not generalizable using simple linear regression model and resulting finer-grained groups using disjoint model.
Table 4.8: Example of application of the Disjoint Model to a group of patterns.

<table>
<thead>
<tr>
<th>Fine-grained pattern groups</th>
<th>Meta-patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diapers = 1, Beers = 0, Chips = 1</td>
<td>Diapers = 1, Beers = 0, Chips = {1, 4}</td>
</tr>
<tr>
<td>Diapers = 1, Beers = 0, Chips = 4</td>
<td>Diapers = 1, Beers = 5, Chips = {4, 7}</td>
</tr>
<tr>
<td>Diapers = 1, Beers = 5, Chips = 4</td>
<td>Diapers = 1, Beers = 5, Chips = {4, 7}</td>
</tr>
<tr>
<td>Diapers = 2, Beers = 3, Chips = 8</td>
<td>Diapers = 2, Beers = 4, Chips = 9</td>
</tr>
<tr>
<td>Diapers = 2, Beers = 4, Chips = 9</td>
<td>Diapers = 2, Beers = 4, Chips = 9</td>
</tr>
</tbody>
</table>

generate the corresponding meta-patterns for that group, leading us to call \texttt{result()}. This function takes as input a set of groups of patterns already finer-grained and correctly separated. It then generates, for each one of those groups, a constant meta-pattern or a disjoint meta-pattern, when the group has only one pattern (line 17) or more than one (line 19), respectively.

Continuing from line 4 (in the case not all bits but one are set yet), we will, for each group of patterns, try to find the best attribute, i.e., the "most decisive one" (line 5) and mark it as used to perform group splitting (line 6). Then, for each pattern in that group, we will separate them taking into the account the values they take considering the previously defined best attribute (lines 8 and 9). This will give us the new groups to work with recursively (lines 11 and 12).

As an example, consider the toy case in Table 4.8. We can see that the optimal separation of patterns is the one represented by the lines in the table. We first separate the patterns by attribute \textit{Diapers} and then \textit{Beers}. The disjoint and constant meta-patterns generated are represented next to each finer-grained group.

We start out by calling \texttt{disjointMetaPatterns()} on the whole group of six patterns. The bitmap is initially empty, which means the condition on line 1 will not be met. We go to line 4 and for each group (the single group of six patterns, in this particular iteration of the algorithm), choose the best attribute. The best attribute will be \textit{Diapers}. This is so because in the group of patterns at hand this is the attribute with fewer different values (only varies between 1 and 2), whereas \textit{Beers} varies between 0, 3, 4 and 5 and \textit{Chips} takes values 1, 4, 7, 8 and 9. Next, we set the first bit of the bitmap to true, since we will use the first attribute (which we determined is the best one) to perform group splitting.

We then iterate each pattern (line 4) with the goal of separating them taking into account the values they take over the best attribute. For this, we use the hash table \texttt{distincts} which will have two keys (one for each value \textit{Diapers} take, i.e., key 1 and key 2). The value for each key will be a list of patterns, the patterns that share a value on the best attribute equal to the key. Every pattern with \textit{Diapers} = 1 will be in the list of key 1 and all patterns with \textit{Diapers} = 2 will be in the list of key 2 (lines 8 and 9). The values for each key (the new lists of patterns) will be the groups to be recursively processed by the algorithm.

We do the first recursive call to \texttt{disjointMetaPatterns()} for the first new group (comprised of patterns that share \textit{Diapers} = 1, i.e., the first 4 patterns of the previous single group). The process is repeated and we now get \textit{Beers} as the best attribute, since although \textit{Diapers} only takes value 1, it has already been used to perform group splitting (as indicated by the bitmap). Hash table \texttt{distincts} will then have key 0 and key 5 (the values Beers take in this new group). We end up with two other groups:
the group of patterns sharing $\text{Diapers} = 1$, $\text{Beers} = 0$, and the group of patterns sharing $\text{Diapers} = 1$, $\text{Beers} = 5$.

Finally, there is a recursive call in which all bits of the bitmap (but one) are set. This means we can generate the corresponding meta-patterns for each group. We generate in this case two disjoint meta-patterns since each group has more than a single pattern.

The same reasoning can be applied to the recursive calls origination from the group with $\text{Diapers} = 2$ (which computation was on hold because of the recursion on the first group). In that case, the only difference would be that constant meta-patterns would have to be generated since in the end we have only groups of single patterns as can be seen in Table 4.8.

4.2.3 Unfolding

When generating the meta-patterns from a set of patterns, the users might be interested in analysing a particular expression that abstracts a specific set of patterns. That meta-pattern might include a large amount of patterns, the expression can be somewhat more complex than the rest, we may be interested in inferring the degree to which the algorithm fails to identify original patterns (or the amount of incorrectly generated patterns from the meta-pattern expression) as a way to evaluate the compression ratio and error trade-off of our algorithm.

All these reasons lead us to formulate what we previously identified as Requirement 6, since we consider important for the user to be able to unfold a meta-pattern in its corresponding patterns, i.e., to generate patterns resulting from a given meta-pattern expression. As an example, one could have meta-pattern $\text{Diapers} = \{1, 2\}$, $\text{Beers} = 2\text{Diapers}$ be unfolded in patterns $\text{Diapers} = 1$, $\text{Beers} = 2$ and $\text{Diapers} = 2$, $\text{Beers} = 4$.

When generalizing a group of patterns, one can create constant meta-items, disjoint meta-items or regression meta-items. The internal representation of constant meta-items is trivial. We simply have to internally store the name of the attribute and the corresponding value it takes for that attribute across all patterns in that group. Regression meta-items are stored in a similar way. In the case of this type of meta-items, however, we have to take into account two variable names (the dependent and the target one), as well as the slope and intercept (refer to Definition 5). As to disjoint meta-items, we could be tempted to further reduce the information stored to only keep distinct values for a given attribute, as that is what is presented to the user, in the final meta-pattern. However, that would not be enough for a correct unfolding of the summarized patterns.

We need to be prepared to deal with the fact that the lossy representation might be used. This fact can lead to ambiguity in terms of what a meta-pattern (i.e., its representation) truly means, as we have previously explained in Section 4.1.1. Also, to avoid all possible combinations of disjoint attribute-value pairs, we have to find a way to generate only the relevant (original) patterns. But at the same time, we need to avoid losing patterns (other than due to the errors introduced by simple linear regression).

Consider that we had the patterns $\text{Diapers} = 1$, $\text{Beers} = 2$, $\text{Chips} = 3$; $\text{Diapers} = 2$, $\text{Beers} = 4$, $\text{Chips} = 3$; $\text{Diapers} = 1$, $\text{Beers} = 2$, $\text{Chips} = 5$ and $\text{Diapers} = 2$, $\text{Beers} = 4$, $\text{Chips} = 5$. We would gen-
**Algorithm 4 Algorithm for unfolding meta-patterns**

**Require:** `metaPattern`

**Ensure:** list of `patterns`

```plaintext
1: patterns ← []
2: for metaItem in metaPattern do
3:   if metaItem.isConstant() then
4:     item ← newItem(metaItem.attribute, metaItem.value)
5:     for p ← 0, metaPattern.originalSize do
6:       patterns[p].add(item)
7:   else if metaItem.isDisjoint() then
8:     for p ← 0, metaPattern.originalSize do
9:       value ← metaPattern.get(metaItem.attribute, p)
10:      item ← newItem(metaItem.attribute, value)
11:     patterns[p].add(item)
12:   else if metaItem.isRegression() then
13:     for p ← 0, metaPattern.originalSize do
14:       value ← metaItem.slope * metaPattern.get(metaItem.target, p) + metaItem.intercept
15:      item ← newItem(metaItem.dependent, value)
16:     patterns[p].add(item)
17:   end if
18: end for
19: return patterns
```

erate, using the lossy representation, the meta-pattern given by `Diapers = {1, 2}`, `Beers = 2Diapers`, `Chips = {3, 5, 6}`.

Imagine we were to only internally store values 1 and 2 to represent the disjoint meta-item for attribute `Diapers`, and value 3, 5 and 6 to represent the disjoint meta-item for attribute `Chips`. We would have to generate all possible combinations during the process of unfolding since we would not know beforehand what were the original patterns. In this particular example, it would lead to the generation of six different patterns, instead of the original four. This problem is aggravated if the number of attributes that originate disjoint meta-items increases. For that reason, we choose to unfold the meta-patterns in a different way.

Our solution is then to keep a more informed internal state of the original settings that lead to the meta-pattern at hand. We choose to keep all possible values for a given disjoint meta-item, even if some redundancy occurs. For example, for the previous example, we would internally store values 1, 2, 1 and 2 for attribute `Diapers` and values 3, 3, 5 and 6 for attribute `Chips`. In the process of unfolding a disjoint meta-item, considering the order of the values follows the order of the patterns, we simply have to, for each disjoint meta-item, get the \(i\)th value, in order to generate the \(i\)th original pattern.

The pseudo-code for the process of unfolding is given in Algorithm 4. We iterate over each meta-item in the meta-pattern and generate items to be added to the different patterns in the list to be returned, considering the type of meta-item we are analysing.

If it is a constant meta-item (lines 3 through 7) we simply have to generate a new item according to its attribute name and value, and replicate it across all patterns. If the meta-item is disjoint, then we need to obtain the value it takes for the given attribute in the \(p\)th pattern (line 10) and generate the item based
on the attribute name and that value (line 11). In case it is a regression meta-item, we need to calculate the value considering the value that the target variable takes for the target variable in the $p^{th}$ pattern, as well as the slope and intercept (line 16).

### 4.3 Extension for Sequential Patterns

We have previously stated that our algorithm is flexible 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 [Barreto and Antunes, 2014] over time series. Time series, which are sequences of data points measured at consecutive equally spaced time intervals [Brockwell and Davis, 2009], 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).

To understand the process that leads to this extension, it is necessary to describe the patterns found using the $TDPrefixGrowth$ algorithm in more detail. Each periodic pattern is composed of an itemset, a set of periods and a duration. The itemset represents the set of items that occur periodically. Periodicity of those occurrences is given by the set of periods, representing the temporal distance between every occurrence of that itemset. The duration is the number of times the itemset is repeated throughout time. The author distinguishes three particular types of patterns: cyclic, converging and diverging. If the periods are equally spaced, i.e., if the itemset occurs always with the same gap (e.g., an itemset always appears after the occurrence of any four other items) then we consider it to be a cyclic pattern. Diverging patterns occur when the periodicity is decreasingly evident (at each time, the frequent itemset occurs with a greater gap, i.e., it tends to disappear). Converging patterns are patterns which itemset occurrence becomes increasingly evident in the course of time.

In the first example of Table 4.9 itemset $ab$ is repeated four times with periodicity 3, 2 and 1 meaning that the occurrence of that itemset becomes increasingly evident in the course of time. From the time series perspective the items would represent values taken by the attribute expressed in that time series (e.g., household energy consumption throughout the day).

As can be observed from the example in Table 4.10, where items are represented as numbers separated by parenthesis, there are still sets of patterns that can be grouped together to form meta-patterns, considering the type of behaviour presented by the periodic patterns. In this case, however, all the values correspond to a unique attribute recorded at different points in time. For this reason, we map

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Pattern</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcdeabigailhab</td>
<td>ab, [3, 2, 1], 4</td>
<td>Converging</td>
</tr>
<tr>
<td>abcabiabghabf</td>
<td>ab, [1, 2, 3], 4</td>
<td>Diverging</td>
</tr>
<tr>
<td>abcdabefabghab</td>
<td>ab, [2, 2, 2], 4</td>
<td>Cyclic</td>
</tr>
</tbody>
</table>

Table 4.9: Examples of sequential patterns found using the $TDPrefixGrowth$ algorithm.
the items in the itemset of each periodic pattern to attribute value pairs, $t_i = x$.

Furthermore, to generate valid meta-patterns, we cannot group together in the same set periodic patterns that do not share the same periods and the same value for the duration, given that those are the main characteristics that differentiate this type of patterns. Otherwise, in the particular case of Table 4.10, we would allow all patterns to be grouped together, which does not translate the real nature of those patterns, ultimately incurring in information loss.

What we can note is that it is the process of grouping that will be changed. Everything else stays the same. The way in which we generalize the itemsets can still be used. Both the Disjoint Model as well as the Simple Linear Regression Model are still valid. In the next sections we explore alternative ways of compaction using these sequential patterns and taking into account the way in which we perform generalizations.

### 4.3.1 Grouping Phase Relaxation

Until now, we have focused our attention on strict rules that must be followed during the grouping phase when talking about periodic patterns. Specifically, we have established that a pattern can only be grouped together with another one if they both share the same itemset size, the same set of periods and the same duration (i.e., if only the values for their attributes vary between each other).

If we are willing to lose a certain degree of accuracy, it can still be useful to understand this type of behaviour for different values of duration (and try to compact those patterns that can in this way be grouped together). The main reason to apply this lossy technique is that if the periods are the same, then the type of behaviour might be evident enough when considering the repetition of a given itemset during “almost the same time” (i.e., approximate duration, given by a variable $\delta$) instead of forcing a strict equality on this parameter. In the example from Table 4.11 one can see that the relaxation used allows a difference of two in the duration (in this case, $\delta = 2$). Furthermore, the resulting meta-pattern is assigned the minimum duration value.

It is important to note that this type of relaxation is only applicable when the set of periods respects the constraint of being equally spaced, meaning that the itemset is repeated in the sequences with a constant gap, i.e., we can only apply this compaction strategy to cyclic patterns. Different durations in

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Meta-patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)(2), [2, 2], 3</td>
<td>$(t_1, 2t_1), [2, 2], 3$ with $t_1 = {1, 2}$</td>
</tr>
<tr>
<td>(2)(4), [2, 2], 3</td>
<td>$(t_2), [2, 2], 3$ with $t_2 = {4, 5}$</td>
</tr>
</tbody>
</table>

Table 4.11: Patterns and corresponding meta-patterns with relaxation on the duration.
4.3.2 Alternative Compaction Strategy

One of the first things we noted when analysing the periodic patterns was that a large number of patterns shared the same itemsets for different periodicities. This fact is most likely due to the nature of the data itself. Considering the domain at hand, it might be logical that a sequence of values of a certain attribute (the one represented by the given time series) happens across different sets of periods. For that reason, we created a way in which the set of periods is the entity subject to mining, instead of the itemsets. Table 4.12 shows an example in which this type of compaction strategy benefits the compaction process.

As can be seen, we generalize the periods, with each attribute taking the same form as time points \((p_1, p_2, ..., p_n)\) and force the patterns that are grouped to share the same itemsets. It is important to note that the algorithm is general enough to enable any grouping strategy to take place by simply defining a custom Grouper which considers that particular grouping strategy (refer to Figure 4.1).

### Table 4.12: Patterns and corresponding meta-patterns when mining periods.

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Meta-patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1)(7), [2, 4], 3)</td>
<td>((1)(7), [p_1, 2p_1], 2 \text{ with } t_1 = {1, 2})</td>
</tr>
</tbody>
</table>

4.4 Optimizations

In this section we present two optimizations that were put into practice as a way to speed up performance of our algorithm and also improve its memory usage. The first one is specific of transactional patterns, that being the reason we focus on this data model when analysing the benefit of these optimizations later in Section 5.4. The second one 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. We now explain each one of them in detail.

#### 4.4.1 Disjoint Meta-patterns

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. This fact is clear in the Case Study chapter (section 5.2). We decided to take advantage of this fact, and developed a data structure capable of benefiting the process of generalization using the Disjoint Model.

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. Also, if patterns are in a group and they share a set of attribute-value pairs, then they might be represented by the same structure, capable of expressing this common part of a set of patterns.
Consider we have, in the Grouping Phase of the algorithm, grouped together the patterns represented in Table 4.13. One can notice that, at the end, we must generate the disjoint meta-patterns \( \text{Diapers} = 1, \text{Beers} = 1, \text{Sodas} = \{3, 4, 5\} \) and \( \text{Diapers} = 1, \text{Beers} = 2, \text{Sodas} = \{1, 2\} \). This means that the first three patterns in the group share the first two items (\( \text{Diapers} = 1, \text{Beers} = 1 \)) and the last two patterns in the group will behave the same way (this time for items \( \text{Diapers} = 1, \text{Beers} = 2 \)). We can further realize all patterns in that group share item \( \text{Diapers} = 1 \). This fact lead us to create the tree-like data structure depicted in Figure 4.3 in which there is a natural sharing of sets of items between different patterns.

In Figure 4.4 we present the basic components that are part of the described data structure. Each one of the circles in Figure 4.3 is what we call a TreeElement. Each TreeElement comprises a set of children nodes and a counter that indicates how many patterns it has considering all its children. This abstract data type can be further specialized into a Root or a Node. The first TreeElement in Figure 4.3 represents a Root in the pattern group. Each group has a single Root, which will have as many children as the number of different values for the most decisive attribute in that group. The Root also comprises the average support of that group and the number of attributes in that group, as well as their names. All other TreeElements in the data structure represented are nodes. A Node is simply a TreeElement composed of an attribute-value pair.

With the data structure defined it is now important to explain some of the most relevant operations that must be performed. We start by explaining the process of construction of the data structure, which is similar to the process of finding disjoint relations (Algorithm 3).

In Algorithm 5, we present the steps to construct the previously explained tree-like data structure from a given pattern group. Starting from the first two lines, we create a new node based on an attribute-value pair (i.e., item), and its number of patterns will be given by the number of patterns in a group (i.e.,
Algorithm 5 Data Structure Construction — construct data structure from a group of patterns

Name: createNode
Require: item, group, bitmap
Ensure: returns node structure for a given item, considering a specific group of patterns
1: node.attribute ← item.attribute
2: node.value ← item.value
3: node.numPatterns ← group.size
4: createChildren(node, group, bitmap)
5: return node

Name: createChildren
Require: node, patterns, bitmap
Ensure: modifies a node to include its children considering a specific group of patterns
6: best ← bestAttribute(patterns, bitmap)
7: bitmap.set(best)
8: distincts ← []
9: for pattern in patterns do
10:   val ← pattern.attribute(best)
11:   distincts.get(val).add(pattern)
12: end for
13: for group in distincts.vals do
14:   item ← getItem(best)
15:   newNode ← createNode(item, group, bitmap)
16:   node.children.add(newNode)
17: end for

the size of that group — line 3). In line 4, we create the children of that node also considering the group of patterns and a bitmap (with the same goal as in Section 4.2.2).

We limit the explanation to the createNode procedure since the createChildren procedure is very similar to Algorithm 3, which we previously referred to. The difference lies in that now we generate a new node for each group of patterns sharing a given value for the n\textsuperscript{th} most decisive attribute, instead of recursively using those groups to promptly generate the underlying disjoint meta-patterns.

As one can verify from lines 4 and 15 these two procedures will call one another, until there are no more items left in the group to create children from. The starting point for the creation of a complete tree for a set of patterns originates from calling createNode with the first item being one of the attribute-value pairs from the most decisive attribute in that group of patterns, that group and a bitmap with a single bit set to true (corresponding to the index of the most decisive attribute).

![Figure 4.4: Main components of the optimized data structure.](image-url)
After having constructed the data structure, we need to have a way to find both types of relations within the patterns in that structure. As we previously mentioned, the idea behind the usage of this data structure is that the Disjoint Model will be the one successfully compacting the patterns found. For that reason, Simple Linear Regression Models were not the priority in this implementation, and thus they will not be as benefited as the former.

We use an extended version of a Weka [Witten and Frank, 1999] class to find the simple linear regression relations between sets of attributes within the group of patterns. In the normal (non-optimized) version of the algorithm we had to iterate through every pattern in the group in order to extract the different values for the items on that group. In the case of the optimized version, this is now equivalent to traversing the tree-like data structure. Because this process is very similar to that of traversing a simple binary tree (although here the data structure is not strictly binary), we spare the reader this explanation and corresponding pseudo-code.

The process of generating disjoint meta-patterns from what are now the groups of patterns (Node elements) is eased. The groups of patterns are built in such a way that constant meta-items need to be created when traversing down the tree and disjoint meta-items will be constructed when the second-to-last node of the tree has more than one children. The pseudo-code for this is presented in Algorithm 6. At each node of the tree, we create the corresponding constant meta-item for that node. If we are in the second-to-last node of the tree, we enter the case in which we might be able to generate a disjoint meta-item (if that node has more than one child — line 4). Otherwise, another constant meta-item is created for the last node (the children of the second-to-last node we are currently analysing — line 6). We then finally add the newly created meta-pattern of this branch to the list of meta-patterns for that Root (i.e., that group of patterns — line 8). In case we are not in the second-to-last node, we simply recursively call the procedure for each child of the current node (lines 10 through 12).

```
Algorithm 6 Disjoint Meta-patterns — construct meta-patterns from the tree-like data structure

Name: disjointMetaPatterns
Require: metaPattern, node, depth
Ensure: modifies meta-patterns list for that node
1: metaPattern.add(constantMetaItem(node.attribute, node.value))
2: if depth = root.numAttributes − 1 then
3:     if children.length >1 then
4:         metaPattern.add(disjointMetaItem(children))
5:     else
6:         metaPattern.add(constantMetaItem(children.first.attribute, children.first.value))
7:     end if
8:     metaPatterns.add(metaPattern)
9: end if
10: for child in children do
11:     disjointMetaPatterns(metaPattern, child, depth + 1)
12: end for
13: end if
```
4.4.2 I/O Performance

Another aspect of our algorithm that could be optimized in the first version, had to do with I/O performance. This was essentially due to the way we were inferring the simple linear regression relations between different attributes within a set of patterns, using the Weka library.

In order to understand this performance bottleneck we resorted to the use of a well-known tool that allows the visualization of detailed information about Java applications while they are running on a Java Virtual Machine (JVM), called VisualVM [Sedlacek and Hurka]. We monitored the Java application that run our algorithm in multiple runs, over transactional patterns. In Figure 4.5 we present an example of the results we got, which is consistent with the different runs of the algorithm. In the figure, it is possible to analyse the hot spots of the Java application that run RECAP. In practice, these are the methods that are more time consuming within the analysed application. We are interested in enhancing the performance of all methods that lie within the d3sm.d2pm.metapatterns package, which is the extension we are applying to the D2PM Project [Antunes].

The most time consuming method is the writeResults one, which cannot be avoided since it is used as part of the Apriori algorithm, but in the scope of our Main file. The second one is setSource which implementation cannot be modified (because it is part of the Weka source code), but may be avoided if we change the process by which we are discovering the simple linear regression regularities. The method generateMetaArff is another candidate for change, since it lines in the scope of our newly created set of classes and which may be reimplemented.

As to the rest of the top 10 hottest methods, we should note that there are three more methods that deal with the process of simple linear regression calculation. They are buildClassifier, writeInstance and (another) setSource. These reasons made us look closer at the process of inferring how one variable connects to the other in some coherent way.

Initially, we were using the Weka library in the standard way. 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 implied that we needed to generate an arff file for each group of patterns so that we subsequently generated the model from it. It is clear that this methodology will implicitly generate a great amount of system calls to deal with the I/O operations of opening, writing, reading and closing the files at hand.

Since our problem is characterized by the dynamic nature of the data, we did not intend to be ob-
ligated to generate a new arff file for each new set of patterns that we found in different runs of the algorithm. For that reason we decided to programmatically (i.e., 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 (an option provided by the Weka library, with specific methods for that purpose). In the end we bypassed all I/O overhead in which we incurred for having to read, write and access files with the information regarding the different groups of patterns.

A comparison that shows the benefits of the optimizations applied is present in the Case Study chapter of this document, particularly in Section 5.4.
Chapter 5

Case Study

In this chapter we apply our proposal and evaluate it taking into account different metrics. The first part of this case study is applied to the education domain and explores the behaviour of our solution when applied to transactional records. In the second part we show that our proposal can also be applied to sequential datasets, in particular we perform analysis in terms of the meta-patterns we find when post-processing the sequential patterns described in Section 4.3. We conclude the case study with an analysis on the performance achieved after the implementation of the optimizations described in Section 4.4.

5.1 Evaluation Criteria

The project proposed in this document must be evaluated taking into account specific metrics that measure its success. In the case of the particular pattern mining subject at hands, it is expected that pattern compression algorithms compress the various sets of patterns found in an advantageous way. Simply put, we need to be capable of extracting what we call a compression ratio which will express to what extent the compaction was successful. The formula we will use to analyse the compaction ratio is given by expression 5.1:

\[
\text{compaction ratio} = \frac{\text{Number of Patterns}}{\text{Number of Meta-Patterns}}
\]  

As an inherently lossy solution, we consider it to be of utmost importance to analyse the error rate of our proposal. Users may be willing to lose some accuracy in the results found, but it will always be limited to a reasonable extent. We consider it is both useful and fundamental to show how our solution behaves in terms of expressing correct patterns using the novel meta-pattern expressions. This leads us to have to analyse how much information might be lost by our approach. There are essentially three situations that can happen at the end of each grouping and corresponding meta-pattern generation: the unfolded patterns originating from the newly created meta-patterns match the original patterns from its group (true positives), some of the unfolded meta-patterns might not exist in the original set of patterns (false positives) or some of the patterns in the group might be left unmatched (false negatives).
last two cases will all happen as the result of the use of the Regression Model, as a consequence of a regression error being allowed to further compact the expressions of patterns found. It is important to note that in the context of our problem, true negatives will not be present, i.e., we are never trying to decide on unfolded patterns that are correctly identified as not being in the set of original patterns (that concept is not useful or replicable in this domain).

Based on the well-known metrics from Information Retrieval, we resort to the concepts of precision and recall to evaluate the outcome of our solution in terms of error introduced and relevance of the meta-patterns found. The equations for those metrics are as follows:

\[\text{precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}\]  
(5.2)

\[\text{recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}\]  
(5.3)

It is clear from the expressions that what we would like to achieve was maximum precision and recall (i.e., 100%). These values will vary taking into account the regression error one is willing to tolerate when generating the meta-patterns for each group of patterns found.

Another important analysis one should take into account has to do with the expressiveness of meta-patterns found. As explained in section 4.1 we will have three types of meta-patterns: regression, disjoint and constant meta-patterns. The optimal solution would find few meta-patterns capable of describing groups of patterns with a single meta-pattern expression. This is not always true, since for various reasons groups often do not allow for the creation of a regression meta-pattern (consequence of the dependence on the regression error provided by the user) leading to finer-grained groups of patterns that will eventually be abstracted by disjoint or constant meta-patterns. We will then analyse the types of meta-patterns generated by our algorithm. We will study how it behaves in regards to this evaluation when submitted to different support values as well as when submitted to different regression errors provided.

Computational complexity in terms of both time and space are also relevant to understand how efficient our solution is. Because of this, we analyse the performance of our algorithm taking into account how time consuming it is and how much memory it uses. Another important aspect to focus when analysing it in terms of processing time has to do with the overhead in which it incurs. Being a post-processing algorithm, the time it takes to apply it to previously mined patterns should not be significant to the point of its use affecting the normal analysis of patterns found by the mining algorithms. Because of that, we compare the performance of our algorithm to that of the mining algorithm used during the mining task.

All time and memory analysis performed are presented as an average of ten runs of each different setting.
5.2 Education

The first set of experimental results we run targets pattern compaction over transactional records. For this part of the case study we have used a private academic dataset to run the experiments. It contains the marks of students enrolled in the Computer Science and Engineering bachelor’s degree spanning approximately ten years of data (from 1990 to 2000) at Instituto Superior Técnico, Universidade de Lisboa in Portugal. While originally containing data about over 6000 enrolled students, the data had to be subjected to pre-processing due to two factors: the heterogeneity in the names of courses that are equivalent throughout different years and the presence of a large amount of course failures.

To solve the first problem, which appears as a consequence of dealing with a decade of data, we normalize the existing courses by merging different but equivalent ones (i.e., courses that have different names but deal with the same topics) into a single course.

The amount of failures is not beneficial for the process of understanding patterns in the data, since there is an unwieldy amount of failures in the data. We initially selected a subset of the subjects which minimized the number of failures. We then selected a number of courses which also contributed to minimizing the presence of missing values. In the process of mining, we do not consider these failures for the process of frequent itemset mining (i.e., failures are not processed by the Apriori algorithm).

This initial pre-processing of the data results in 2614 records. Each one of the 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. 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 an approximation to the American grading system, here translated from letters to numeric values (necessary for our algorithm).

All the experiments regarding this dataset were performed in 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 [Antunes] to collect the frequent itemsets that are further post-processed by our algorithm.

5.2.1 Compaction Ratio

We start the analysis of our experimental results by trying to understand how the value of $\Delta \sigma$ contributes to the process of meta-pattern inference and how it limits the possibilities of generalization. Results of running the algorithm in the educational dataset considering different support values and while varying the grouping variable $\Delta \sigma$ between 5\% and 999\% with regression error fixed at 2 are shown in Figure 5.1. The amount of meta-patterns reported (which are generated in the **Generation Phase** of our algorithm) are inherently dependent on the groups created in the **Grouping Phase**. The larger the amount of patterns, the more patterns will be candidate to be abstracted into meta-patterns. With the decrease of
restrictiveness in terms of support values, it is only natural that more broader groups (groups including more patterns) are created, thus leading to better generalization processes to occur. As we can see from Figure 5.1, the amount of meta-patterns decreases substantially from using $\Delta \sigma = 5\%$ to about half of that amount when $\Delta \sigma = 999\%$.

The results in terms of numbers of meta-patterns reported to the user from running the compaction algorithm with minimum support set to 10%, 7.5%, 5% and 2.5% are shown in Figure 5.2. Note that in these tests we allow permissive pattern grouping in the Grouping Phase of the algorithm, making $\Delta \sigma$ virtually infinite (in fact, $\Delta \sigma = 999$), which means we will group patterns that share the same attributes disregarding the support values, attributing an overall support which is the average of the supports in the grouped patterns, as explained in section 4.2.1. In the figure presented, we show results considering the original amount of patterns, the amount of meta-patterns generated using the lossy representation we discussed in Section 4.1.1 with and without regression error for the marks (set to 4 in this case, which corresponds to tolerating an error of up to two American marks different from the original result — an error which is arguably too permissive). Furthermore, we present results of the amount of patterns considering the lossless representation with the same regression error as before. We also run the Eclat algorithm [Zaki et al., 1997] using the openly available implementation by [Borgelt, 2003] to assess the amount of patterns reported to the end-user when using closed and maximal sets. As one can see, the
closed patterns take no effect on reducing the number of patterns found without missing values (note the overlap between patterns and closed). This can naturally be attributed to the restrictive nature of this algorithm (a lossless algorithm), associated with the fact that in the case of this particular dataset the supports of patterns will be often times different between frequent itemsets and corresponding supersets (otherwise the amount of closed itemsets would be less than the values reported). 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 when using the maximal set strategy. We have to understand that the permissiveness of meta-patterns contributes highly to the degree of generalization it achieves. This is not so without loss of accuracy, as we will show next. What is also clear is that in the dataset at hand, associated with the regression error provided, the difference between using the lossy and the lossless representation strategy incurs in more or less the same amount of meta-patterns. However, as would be expected, the amount of meta-patterns is further reduced when we allow the lossy representation to take place.

In Figure 5.3, we present the compaction ratios for different values of regression error using the lossy representation version of our algorithm (using the computed value according to Equation 5.1). As was expected from Figure 5.2, and from the way the algorithms works, the greater the regression error, the larger the compaction. But what is interesting to analyse is how precision and recall, identified in section 5.1, behave when we vary the value of the regression error and how evident they become with those variations. For that, we are forced to use the process of unfolding the meta-patterns that are created by our algorithm and then match each of the patterns with the original ones.

5.2.2 Precision and Recall

As previously explained, what may happen is that we end with more patterns on the unfolded meta-patterns or original patterns are left unmatched due to the allowance of a regression error when computing simple linear regression. In Figures 5.4a and 5.4b we show how precision and recall vary for different support values and considering regression errors set to 2 and 4, respectively. Note that the chart for precision and recall considering regression error of 0 is not present because it would consist simply of a straight line hitting 100% for both metrics, since we allow no error in the regression, thus limiting
the generalization but enhancing precision during the process of unfolding the meta-patterns that were generated.

First, we can observe that the values for precision are always higher than 80%, which means that the RECAP algorithm is able to return a substantial number of correct patterns (i.e., it gets right most of the patterns identified as existing ones). The values for recall are always smaller than the values for precision, which can lead us to conclude that the algorithm fails to return some of the meta-patterns that should translate to original patterns.

It is also important to note that contrary to the general case when using precision and recall, here there is no relationship of inverse proportionality between these two metrics. This fact can be observed in both charts of Figure 5.4. Contrary to a classification or information retrieval problem, here we do not measure precision or recall taking into account correctly classified instances or relevant documents, but instead correctly generated (i.e., unfolded) patterns. What we can conclude is that in the case of meta-pattern generation, when increasing the number of false positives, the number of false negatives generally increases at a higher rate. Thus, when submitted to a regression error $\varepsilon > 0$ the RECAP algorithm will contribute more to misidentifying previously existent (original) patterns but will not generate that many patterns that did not exist.

In order to better understand this counter-intuitive behaviour of our algorithm, we now get into some of the details of why this happens, i.e., what justifies the fact that precision and recall are not inversely proportional. In Figure 5.5 we illustrate the relation between false negatives, false positives, true positives and incorrectly misidentified patterns, incorrectly identified patterns and correctly generated patterns, respectively.

The questions here lie in two distinct points. The first is the reason why recall and precision do not follow the proportionality trend. After understanding why this happens, we must also have clues as to why recall is consistently worst than precision in the scope of our problem.

To answer these questions we introduce Figure 5.6. In this example, we can see that from the four original patterns, considering a certain degree of regression error, we might generate the meta-pattern in the middle, which will lead to the unfolded patterns on the right (only three).

The reason recall and precision are not inversely proportional has to do with the way false negatives and false positives are generated in our domain. If we analyse Figure 5.6 (particularly the correspondence between the last patterns in each side) you can see that whenever we generate a false positive
(Diapers = 3, Beers = 6), a corresponding original pattern is left unmatched (false negative Diapers = 3, Beers = 5), because the false positive occurs as an approximation of an original pattern. In this way, an increase in false positives will also lead to an increase in false negatives, having a direct impact in the trends followed by precision and recall, respectively.

To generate the meta-pattern we introduce a regression error. This leads to two consequences. The first one is approximation (which is evident in the case of original pattern Diapers = 3, Beers = 5 that became the unfolded pattern Diapers = 3, Beers = 6). But it also introduced what we might call "merge", e.g., the original patterns Diapers = 1, Beers = 2 and Diapers = 1, Beers = 3, both became "represented by" (were merged into) the single pattern Diapers = 1, Beers = 2.

In Figure 5.6, true positives are identified by blue rectangles (Diapers = 1, Beers = 2 and Diapers = 2, Beers = 4). The rest of the patterns will then correspond to false negatives or false positives if they are in the original patterns or in the unfolded patterns side, respectively.

Recall is always smaller than precision, which means we usually have a larger number of false negatives than we have false positives. It is then important to know what causes false negatives and false positives to arise. The answer to that question is straight forward from the previous explanation. We introduced a false negative when we merged the top two original patterns (Diapers = 1, Beers = 2 and Diapers = 1, Beers = 3) into the first single unfolded pattern on the right (Diapers = 1, Beers = 2). But we also introduced a false negative when we failed to represent the last pattern on the left (Diapers = 3, Beers = 5). False negatives then appear as a consequence of two things: merge and approximation (the two cases we just described). False positives, however, can happen only due to one of those reasons. The only false negative we see appearing results from the fact that there is a newly generated pattern that has no correspondence on the original patterns side (Diapers = 3, Beers = 6), i.e., it happens simply as a consequence of approximation.

It is then clear that recall will suffer a greater impact in the process of meta-pattern generation, since in the specific domain of our problem the number of false negatives introduced will be influenced by two

![Figure 5.5: False negatives, false positives and true positives Venn diagram.](image-url)
Continuing with the analysis of the experimental results, as can be seen from Figure 5.3, the best compression ratio reaches almost 3, for minimum support threshold of 5\% and using a regression error of 4. It is also in this set of conditions that we get the worst results in terms of both precision and recall — about 67.5\% recall and 80\% precision. From the results in Figure 5.4b we get the idea that this is an overly permissive strategy, particularly considering the regression error that we are using in that experimental run.

In Figure 5.7, we present a different representation of the values shown in Figure 5.4, but now limited to minimum support values of 2.5\% and 7.5\%. It is noticeable that the increase of regression error is accompanied by a decrease in precision and recall.

There is one interesting case that happens when the minimum support is set to 2.5\% for \( \varepsilon = 2 \). In this case, the compression ratio is higher than that of the same experiment run with minimum support 7.5\%. But, contrary to the direct relation we had identified before, precision and recall are higher for minimum support 2.5\% (with a higher rate of compaction) than when the minimum support is 7.5\%. This most likely happens because even though a higher compression occurs, it must be due to the amount of disjoint meta-patterns found and not regression meta-patterns (the ones contributing to the lossy side of our approach). It thus becomes important to analyse the relative amount of types of meta-patterns found in each case.

### 5.2.3 Types of Meta-Patterns

Figures 5.8a and 5.8b show the distribution of types of meta-patterns found for each one of the regression errors considered. We can observe that the increase on regression meta-patterns is coherent with the decrease of precision and recall. There is a slight growth in the amount of regression meta-patterns found from regression error \( \varepsilon = 2 \) to \( \varepsilon = 4 \). It is also important to note the absence of errors (i.e., precision and recall reach 100\%) in both cases for minimum support \( \text{minsup} = 10\% \). In these cases, no regression meta-patterns were created by the RECAP algorithm.

With the previous analysis, one might be misled to think that one is only able to find regression meta-patterns when allowing some sort of regression error. Of course this aspect is very data dependent, but we can find cases in the educational dataset we are using that prove the existence of simple linear regression relations between attributes even when we allow no regression error. The analysis of types of meta-patterns for regression error \( \varepsilon = 0 \) is depicted in 5.9.
5.2.4 Performance

In Figure 5.10, we present an empirical analysis of the performance of our algorithm both in terms of time and space. We run the experimental results considering different regression error values, permissive support grouping and lossy representation.

As can be observed from Figure 5.10a, memory usage increases at an exponential rate in our algorithm, with the decrease of the minimum support value. This behaviour is natural and expected, since we are post-processing patterns that were the result of a mining task that will generally generate exponentially increasing patterns for growing minimum support values. We can observe that memory reaches a maximum of around 56MB when submitted to the lowest minimum support value ($\text{minsup} = 2.5\%$) when allowing no regression error. This comes as a direct consequence of the data structures used in the process of creating the meta-patterns capable of describing the different groups of patterns, and also because of the permissiveness of the regression error. As can be seen in the same figure, a more lossy compaction strategy with $\varepsilon = 4$ incurs in 40MB, a reduction of about $30\%$ in memory usage.

In terms of computational processing time, we can see in Figure 5.10b that the behaviour is similar to the spatial chart (but in this case the progression is less pronounced). The similar behaviour can be explained by applying the same reasoning as in the memory usage analysis. While post-processing an exponentially increasing amount of patterns, we will also spend more time performing grouping and generalization in the two separate phases of the algorithm, ultimately contributing to an increase in the overall processing time. The time spent between different runs of the algorithm can be more easily iden-
tified in each minimum support value than in the spatial case, showing that the difference on regression error will impact time stronger than memory for a fixed value of minimum support threshold.

The trade-off of compaction, memory consumption and algorithmic performance versus information loss becomes even more clear with the previous analysis. But we also consider important to look closer to the impact our proposal has as a post-processing algorithm. For that, we resort to another time analysis, by comparing it to the time spent by the Apriori algorithm implemented in the D2PM project. The results are shown in Figure 5.11. In the figure, we can see that for different values of regression error, the overhead in which our algorithm incurs after Apriori has finished can be seen as negligible in the first three cases of minimum support. The worst result we get happens when no regression error is present and the minimum support threshold is set at 2.5%. And even with this worst result, the algorithm incurs in less than 30% of the processing time if we consider the total time spent by both algorithms together. We consider this to be a fairly good result, that does not contribute with a significant overhead.
after the normal processing time. Thus, we believe such a task can be seen as non-prejudicial in terms of time spent for the process of inferring and presenting the user with a compact version of the patterns found by the mining task.

### 5.3 Power Consumption and Healthcare

For the second set of experimental results we target pattern compaction over sequential records. We study the different compaction strategies we introduced in Section 4.3 that are applicable to the case of the particular type of sequential patterns at hand, that are the result of running the TDPrefixGrowth algorithm. For this part of the case study we have used two datasets publicly available in the UCI Repository [Bache and Lichman, 2013]. The first one, the Household Power Dataset (HHP), refers to measures regarding expended power (in kilowatt) at different temporal instants, with the dataset being pre-processed in order to display the measures at day granularity. We have used 1431 daily records (from 2006 through 2010 — about 5 years of data) with each one of the records comprising 1440 measures, which corresponds to one measure per minute for each day. Furthermore, the measures form an alphabet of 101 different values. The second dataset, in the healthcare domain, refers to electroencephalogram measures (EEG) from a total of 87 different patients and multiple sensors. This dataset was also pre-processed. We have normalized the measures from different sensors, enabling its analysis of the variation of the values independently from its scale. With this, we have used a total of 5568 records (from all the sensors of the 87 patients) that comprise a total of 256 contiguous measures each one. The measures vary between 11 different values.

#### 5.3.1 Cyclic Patterns

We begin by analysing cyclic sequential patterns and run the algorithm with different minimum support values for distinct compaction strategies. Similar to the study we performed in the first dataset, we start by studying the effect of the presence or absence of the lossless representation in the tested dataset. Next, we study the compaction ratio considering the information presented to the user before and after our algorithm post-processes the patterns. We also perform performance tests, as we are interested in understanding how the algorithm will behave in terms of time complexity in the case of sequential patterns. We make the same overhead analysis of the RECAP algorithm as a post-processing algorithm as before, now applied to the domain of sequential data. An analysis on information loss is also performed, in which we calculate the amount of patterns that could be restored from the inferred meta-patterns.

#### Compaction Ratio

We start by showing the experimental results regarding the cyclic patterns found in the HHP dataset. In Figure 5.12a we present the results of applying the RECAP algorithm considering different configurations and options potentially provided by the user for different values of minimum support. We show results in
terms of the amount of meta-patterns found against the original amount. These meta-patterns can be
found using the lossy or lossless expressions for their representation (refer to Requirement 5), the user
may allow a certain degree of error in the simple linear regression or even tolerate an error in terms of the
duration of the cycles (as explained in Section Grouping Phase Relaxation). As verified by Figure 5.12a,
the amount of information presented to the end-user decreases drastically by simply using the lossless
representation of meta-patterns considering no regression error and no relaxation on the duration value
of the cyclic patterns — in the most visible case, with \( \text{minsup} = 1\% \) we start with an original amount of
about 2250 patterns and we are capable of finding regularities between them such that we are able to
generate less than half of that value in meta-patterns, represented by the green solid line in the chart
(less than 1000 corresponding meta-patterns capable of correctly describing the patterns found). It is
also possible to understand from Figure 5.12a that the different variations of settings in the RECAP
algorithm are only noticeable when the minimum support value is set to 1\%. For that reason, we will look
closer to what happens in that scenario.

From Figure 5.12b we can see that the difference between using the lossy or the lossless representations has a much higher impact in this dataset that in the previous one (refer to Figure 5.2). This is not so much due to the nature of the patterns, i.e., it is not due to the fact that we are processing sequential patterns instead of transactional ones. It is rather because the number of items in each pattern is much higher than that of the transactional dataset. In the education dataset the largest patterns found were composed by four attribute-value pairs. In the case of the household power consumption this is not so, as we have patterns reaching 19 different attributes (for \( \text{minsup} = 1\% \)), i.e., a set of 19 consecutive values is repeated with a cyclic behaviour in this dataset. If we remember the explanation of Requirement 5 and Table 4.2, we understand that the higher the number of attributes in the patterns, the lower the probability of being able to generate regression meta-patterns with the lossless type of compaction.

**Types of Meta-Patterns**

As a way to analyse the types of meta-patterns found by our algorithm when considering the HHP
dataset, we fix the minimum support value at 1\%. First, we focus on the lossy compaction strategy
depicted in Figure 5.13a (which in the case of this particular dataset is overly optimistic). This comes as
a consequence of the types of meta-patterns that are constructed if we allow the lossy representation.
In case we find one regression meta-item, then the other attributes will be candidates to be disjoint meta-items, and this will eventually lead to the existence of a large number of regression meta-patterns (which, following Definition 8 is any meta-pattern that includes a regression meta-item). As we previously explained in Section 4.1.1 it is important to note that this only affects the expression initially presented to the user, but not the correctness of the corresponding unfolded patterns, i.e., if the user unfolds the meta-patterns found, the algorithm would not generate all possible combinations inherent to that lossy expression. We consider that the lossy expressions, contrary to the previous dataset, do not benefit the understanding of the patterns found. We now present an analysis on compaction considering the lossless representation, which we find to be more useful in the case of this dataset.

In Figure 5.13b we can observe that in this dataset, both regression error and relaxed duration do not contribute to a great extent on improving generalization in regards to cyclic patterns. This can be verified by the small percentage of regression meta-patterns that we are able to find for these values of regression error and relaxed duration when compared to the situation in which we allow no regression error or relaxation in terms of duration of the cyclic patterns. In these experimental results, the regression error used was $\varepsilon = 0.1$, which we believe is a reasonable enough degree of error considering that values range in the tenth of units, i.e., attributes take values like 0.1, 0.2, ..., 1.1, and so on. The relaxed duration variable was set to 2, which means we allow cyclic patterns to be grouped together even if they are off by 2 units in the value for the duration.

**Precision and Recall**

In Figure 5.14 we show values for precision and recall considering the different strategies we have used for cyclic patterns compaction, for minimum support again fixed at 1%. We can observe that with the increase of permissiveness in both grouping (using the relaxed duration compaction strategy for cyclic patterns) or in simple linear regression error, the precision will decrease, as expected. Recall follows the same trend, although more pronounced, since as we saw in the transactional dataset regression meta-patterns will contribute more to misidentifying original patterns. The most interesting analysis comes from the fact that in the case of this dataset we have very little error when using the regression error strategy without relaxed duration ($\varepsilon = 0.1, \delta = 0$). On the other hand, if we allow relaxed duration and tolerate no error in the simple linear regression, we get a very different result (as shown by column $\varepsilon = 0, \delta = 2$). It is useful to analyse why this happens and also to understand to what extent this

![Figure 5.13: Types of meta-patterns found for cyclic patterns over the HHP dataset.](image)
increase in percentages of errors contributes to the compaction process. The reason for this to happen is straightforward. By allowing a relaxation in terms of duration of the cyclic patterns, we will have a significant number of patterns that will be looked at as the same one. Imagine we had cyclic patterns comprising the same itemset that only had different durations, but all with a duration offset of 2 between each one. What will happen in the relax duration compaction strategy is that they will all be allowed to be grouped together, and hence repeated itemsets will be discarded. This contributes to a great decrease in recall as false negatives will arise, since we will now only generate one of the originally patterns found for that “merging process” that inherently happens in the grouping phase when we relax the duration.

If we analyse Figure 5.14 also taking into account Figure 5.12b, we can conclude that the trade-off between compaction and errors is much better when using just the simple linear regression error set at $\varepsilon = 0.1$. Also as expected, when we allow no error in the grouping or generalization, precision and recall will be set to 100%, and when the strategy used is the most permissive one (allowing both regression error as well as relaxed duration), we get the worst results for both metrics.

From the previous experimental results it is clear that the relaxed duration strategy is not that useful in the particular case of this dataset. However, this compaction strategy is very dependent on the data itself. For that reason, and in order to try to show the usefulness of this compaction strategy, we test the same set of configurations in the EEG dataset. It makes sense to use the same configurations because the values of the EEG dataset also vary in the tenth of a unit (hence we keep $\varepsilon = 0.1$). We also choose to keep the same value for the relaxed duration $\delta = 2$. In Figure 5.15a we present the amount of information reported to the user for the case of the EEG dataset, with minimum support fixed at 1%, for different compaction strategies. We also present the corresponding information loss (in the form of precision and recall) in Figure 5.15b. We can conclude that in this dataset, the relaxed duration compaction strategy is useful in further compacting the patterns found by the TDPrefixGrowth algorithm. We can see that regression error does not contribute to the process of compaction in this case, which will be reflected in the precision and recall for $\varepsilon = 0.1, \delta = 0$. Recall drops to about 55% in the case of EEG with $\varepsilon = 0, \delta = 2$, for the same reasons explained before for the relaxed duration compaction strategy in HHP. The grouping phase is the only stage of the algorithm responsible for information loss,

![Figure 5.14: Precision and recall considering different compaction strategies for cyclic patterns over the HHP dataset.](image-url)
5.3.2 Converging Patterns

Until now we have focused our attention in the sequential cyclic patterns. However, the result of applying the algorithm to converging and diverging patterns must also be analysed, particularly taking into account the different compaction strategies we presented in Section 4.3. We start by performing such analysis on the HHP dataset.

For the same reasons as in the case of cyclic patterns, we analyse compaction considering only lossless expressions. We also resort to an analysis only at the level of converging patterns, since the nature of the remaining two types of sequential patterns (converging and diverging) is very similar, the only difference being that in converging patterns the algorithm considers increasingly evident patterns whereas in diverging it will look for decreasingly evident ones, but the characteristics of these patterns are the same.

In Figure 5.16a, we show a comparison on the amount of information shown to the user using the different compaction strategies available for converging patterns, considering different values for minimum support. When using the standard compaction strategy (itemsets compaction) with regression error, we set it to 0.1 ($\varepsilon = 0.1$). When applying the alternative compaction strategy for converging and diverging
patterns (periods compaction) with regression error, we set it to 1 \( (\epsilon = 1) \), only allowing an error of 1 in terms of the periods of a given set of patterns).

From the figure, it is clear that in the case of converging patterns for the HHP dataset, compaction only varies significantly with minimum support set to \( \text{minsup} = 1\% \). For the other values of minimum support we are not able to find a considerable amount of regression meta-patterns and that fact will lead to a constant behaviour considering the different compaction strategies, i.e., we are not able to further generalize the patterns found. We can see that the two different strategies for converging and diverging patterns (itemset and period compaction) happen to compact the patterns to about the same extent for values of minimum support larger than 1\%.

As before, in Figure 5.16b we focus on the compaction achieved for minimum support \( \text{minsup} = 1\% \), as it will allow us to analyse what compaction strategies work best in each dataset. In the case of HHP we can see that period compaction benefits the compaction process in detriment of the "normal" itemset compaction. This means that it is more frequent to find patterns that share the same itemset but vary in terms of when they occur. In this way, the algorithm is able to more broadly generalize patterns by periodicity rather than by the different values they take for the same set of periods and duration.

Figure 5.17a shows the types of meta-patterns found considering different compaction strategies. In the particular case of converging patterns over the HHP dataset, we get very good results. As can be seen, the amount of regression meta-patterns found for each compaction strategy is relevant enough (particularly when period compaction is used). At the same time, we are able to achieve a very small amount of errors (note the scale in the vertical axis of Figure 5.17b). It is particularly interesting to observe that when using period compaction without regression error (third column), we reach about 15\% of regression meta-patterns. If we allow an error of 1 in the periods, we increase this number to about 20\%, with negligible error in terms of the generated patterns from the meta-patterns found (last column).

In Figure 5.18 we compare the results obtained for the same minimum support and configurations over the different compaction strategies for the case of the EEG dataset. What we realize from Figure 5.18a is that in the case of this dataset the amount of regression meta-patterns that can be inferred from the original set of converging patterns is very limited in the case of the itemset compaction strategy and effectively inexistent in the case of period compaction. As a consequence, the error is minimal (Figure 5.18b). This is the worst result we get in terms of diversity of meta-patterns found, although we still have a considerable reduction in the amount of information presented to the user, as can be verified in Figure

![Figure 5.17: Types of meta-patterns and corresponding precision/recall for converging patterns over the HHP dataset (minsup = 1%).](image-url)
5.3.3 Performance over Sequential Patterns

We now analyse performance in terms of time and space in regards to both cyclic and converging patterns from the HHP dataset, considering different values of minimum support and different compaction strategies.

Starting from Figure 5.20, we fix minimum support at $1\%$ and analyse both cyclic and converging meta-patterns found in the HHP dataset in terms of the time spent by the RECAP algorithm and time spent by TDPrefixGrowth. We can see that, just like in the case of transactional patterns, the time spent by our algorithm is negligible when compared to the process of mining the patterns from the original dataset. These experimental results allow us to state that the task of meta-pattern compaction in the case of sequential patterns may be applied without significantly harming the process of finding patterns in the data.

In Figure 5.21 we present results in terms of time consumption, considering different compaction strategies for cyclic and converging patterns, and different minimum support values. This allows us to empirically analyse the time complexity of our algorithm. What we can see from Figure 5.21a is that there is an almost constant behaviour of our algorithm in regards to cyclic patterns for the first
values of minimum support, with a significant spike in the value of seconds spent for minimum support $minsup = 1\%$. The seemingly constant behaviour is explained by the low amount of time consumed with minimum supports set to less than $1\%$. The peek at $1\%$ happens due to the explosion of patterns that occurs when we set the algorithm to the lowest minimum support value, thus making it so that the RECAP algorithm has a greater amount of patterns to process (one can verify this variation in amount of patterns reported by TDPrefixGrowth in Figure 5.12a). In Figure 5.21b, on the other hand, it is noticeable that the period compaction strategy for converging patterns is generally slower than the itemset compaction one. The tendency, however, is the same for cyclic and converging patterns. In both cases we can conclude a general exponential behaviour in time consumed by the algorithm, as expected. The lower the support, the greater the amount of patterns found (at an exponential rate, since subsets and supersets of the patterns found are reported to the user using the TDPrefixGrowth algorithm, as when using Apriori for the transactional case).

Figure 5.22 shows the evolution in terms of memory consumption for different compaction strategies again considering the cyclic and converging meta-patterns found in the HHP dataset. As we had seen before in the case of transactional patterns, memory increases exponentially with the decrease of minimum support. This exponential behaviour is more clear in the case of converging patterns, with cyclic patterns (like in the time consumption analysis) having an almost constant growth for minimum supports greater than $1\%$. This fact is most likely justified by the considerable greater amount of converging patterns found when compared to cyclic ones over the same dataset, thus being more noticeable in the later.
5.4 Optimizations

We test the optimizations explained in Section 4.4 over the transactional dataset (education domain). In Figure 5.23a we can compare performance of the RECAP algorithm in its unoptimized version with the optimized version in which we are using the data structure targeted at enhancing the process of disjoint meta-patterns creation and also avoiding I/O overhead. We test performance over different minimum support values, considering the lossy representation while tolerating no regression error. Permissive support grouping is used, with $\Delta \sigma$ virtually infinite.

As can be observed, performance is much better if we use the optimized version of the algorithm. In fact, it is evident that the optimizations are useful at any minimum support value, even when it is set to the maximum value we consider ($\minsup = 10\%$). Although the behaviour is still exponential — which is comprehensible, as the changes made did not allow for the asymptotic computational complexity to change since we did not modify the nature of the essential parts responsible for the performance overhead of the algorithm (mainly the complexity incurring from the number of patterns having to be analysed and the groups that are formed) —, the growth rate of the time function in regards to the decrease of the minimum support value decreased significantly, now being less pronounced.

In order to test the impact of our optimized version (particularly considering the data structure used), we perform memory tests in which we analyse the heap usage throughout the running time of the algorithm. What we expect is that memory usage decreases from the non-optimized version to the optimized one, since in the later we do not need to keep all patterns in memory as they are, since we will be able to share a considerable amount of information between patterns in the same group, as explained in Section 4.4.1. In Figure 5.23b we present the result of running the RECAP algorithm.

Figure 5.23: Comparison between the non-optimized and optimized versions of the RECAP algorithm.
with the same set of configurations as in the time analysis, but this time with minimum support set at \( \text{minsup} = 1\% \). As we can observe, the amount of memory used decreases from the non-optimized to the optimized version of the algorithm. However, it is not as beneficial as we initially thought it could be. This is most likely due to the fact that even though in theory the structure shares a relevant amount of information between different patterns, the Java data structures we are using are probably incurring in a great overhead, leading to a situation in which the normal pattern groups and pattern objects end up using as much space as the alternative data structure used in the optimized version of the algorithm.

It is interesting to observe that the rate of creation of objects (directly expressed by heap usage) increases at a faster rate in the optimized version than in the non-optimized one. This is most likely due to the way we generate the Root objects in the optimized version. We are creating them for each group of patterns resulting from the grouping phase, which will cause groups of patterns to exist in parallel with roots for already abstracted groups (using the tree-like data structure).

In the end, we have an optimized version that is both faster and less space consuming than the initial version of the RECAP algorithm, with time performance being the one that benefits the most from the optimization.
Chapter 6

Conclusions

The explosion of the number of patterns discovered by frequent itemset mining algorithms, along with the difficulty on analysing such number of patterns, have impaired the generalized use of pattern mining in data analysis. In this thesis, we proposed a method for compacting the discovered patterns into a more manageable set, creating abstractions of the original patterns discovered — the meta-patterns. We now do a brief explanation of the major achievements of this work and hint for possible directions of future work.

6.1 Achievements

In this work, 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 significant, and the time spent on this step is negligible compared to the time spent during the mining task. The errors introduced with the permissiveness measures of our algorithm do not impact the quality of the meta-patterns found to a great extent, as we are able to generally achieve both good values for precision and recall, when comparing the unfolded patterns with the original set of patterns found.

We also explored the idea of applying our proposal to the problem of sequential pattern mining, by finding regularities across sets of time frames, allowing us to discover dependencies between the attributes of patterns expressing frequent behaviours across different moments in time. Ultimately, we compact sequential patterns using the same methodology as before, with some differences in terms of the grouping phase of our algorithm. Furthermore, we proposed and applied alternative compaction strategies to be used in the case of sequential patterns. One of those approaches — periods compaction — is shown to benefit the results of the compaction process.

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 the frequent itemsets. If the process of generalization was done in runtime, i.e., while analysing the itemsets in the database, we are confident better performance times could be
achieved. On the other hand, an advantage of this type of algorithm is that it can be applied on top of other mining approaches that might consider different criteria when mining the datasets.

6.2 Future Work

Although we have showed that simple linear regression can be useful in reducing the amount of information that is presented to the user, we believe new types of meta-items might be created besides the presented regression meta-items. In that way, we might be able to model polynomial or exponential growth factors on variables. Disjoint meta-items might also be extended to deal with ranges. In a large number of cases, the values of the disjoint meta-items found were comprised of consecutive numbers. A range could then be used to further reduce the amount of data shown to the user.

The properties of the patterns grouped together are related to the concepts of additive and multiplicative models that are referenced in some biclustering algorithms [Madeira and Oliveira, 2004] (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. This would have an impact in performance, since the groups resulting from different sets of patterns would lead to a combinatorial explosion in the search-space.
Bibliography


C. Antunes. Project d2pm. [https://sites.google.com/site/projectd2pm/](https://sites.google.com/site/projectd2pm/).


