UNIVERSITEIT ANTWERPEN
Departement Wiskunde en Informatica
Academiejaar 2005-2006

Association Rule Mining met Missing Values

Michael Mampaey

Proefschrift ter verkrijging van de graad van Licentiaat
in de Wetenschappen, richting Wiskunde-Informatica
Promotors:
Dr. Toon Calders
Dr. Bart Goethals

Acknowledgments

I would like to thank my promotors Toon Calders and Bart Goethals for their advice, guidance and suggestions in the process leading to the completion of this thesis.

Contents

List of Figures

Nederlandstalige Samenvatting

Missing values vormen een belangrijk en onvermijdelijk probleem in Data Mining, en in data management en analyse in het algemeen. De gebruikelijke technieken kunnen niet rechtstreeks met missingness werken, en vereisen op zijn minst één of andere vorm van pre-processing, waarbij noodgedwongen bepaalde veronderstellingen gemaakt moeten worden. In deze thesis geef ik een effectieve oplossing voor dit probleem, evenals een efficiënte implementatie in een algoritme, XMiner, dat inherent association rules kan minen in databases met missing values.

In het Association Rule Mining (ARM) model, is het doel het vinden van alle regels van de vorm $X \Rightarrow Y$ in een transactionele database $\mathcal{D}$, waarbij X en Y sets van items zijn, die intuïtief gezien, frequent voorkomen en een hoge probabiliteit hebben. Om de kwaliteit van dergelijke regels te kunnen beoordelen, bestaan er twee maten *support* en *confidence*. De meeste ARM algoritmen kunnen in twee stappen opgedeeld worden: frequent set mining en rule generation. Eerst worden alle itemsets met een hoge support gezocht, waarmee dan de association rules met hoge confidence gegenereerd kunnen worden. Deze laatste stap is vrij eenvoudig. Bij frequent itemset mining wordt de exponentieel grote search space over alle itemsets doorkruist (deze vormen een subset lattice), gebruik makend van de monotoniciteitseigenschap, of de Apriori property. Deze eenvoudige maar fundamentele eigenschap zegt dat een infrequente itemset zelf geen frequente supersets kan hebben. Dit laat toe om grote sublattices met een infrequente itemset als root volledig te kunnen verwijderen uit de search space. Het doorkruisen van de lattice kan breadth-first of depth-first gebeuren, er bestaan vele gekende voorbeelden waaronder Apriori [2], Eclat [14] en FP-Growth [6]. Oorspronkelijk is association rule mining gedefinieerd voor transactionele databases, maar het is gemakkelijk om over te schakelen naar het relationele database model (alhoewel dit niet geheel vanzelfsprekend is). Aangezien relationele databases meer realistisch zijn, en het probleem met missing values er natuurlijker voorkomt, zal deze thesis ook voornamelijk hiermee werken.

De introductie van missing values in een database brengt vele problemen met zich mee voor een ARM algoritme. Support en confidence worden vervormd, en dit kan leiden tot het verlies van goede of juist het verzinnen van slechte regels. De oorzaak van de missingness moet hierbij beschouwd worden. Ontbreken er waarden volledig willekeurig? Kunnen nulls als aparte waarden behandeld worden? Kunnen er nuttige afschattingen gemaakt worden voor de ontbrekende waarden? De gebruikelijke aanpak is om ofwel gegevens toe te voegen, door bijvoorbeeld een gemiddelde attribuutwaarde te gebruiken, ofwel tupels die nulls bevatten simpelweg te verwijderen uit de dataset. Het is duidelijk dat deze methodes de distributie van de data grondig kunnen verstoren, wat een nefast effect kan hebben op de output.

In dit werk worden de support en confidence maten op een backwards compatible manier geherdefinieerd, samen met een nieuwe maat *representativity*, die niet-missingness of waarneming uitdrukt. Deze definities komen uit voorafgaand werk door *Ragel en Crémillieux* [8]. De nadruk in hun werk ligt op de eigenschappen en de kwaliteit van deze nieuwe definities. Ze tonen bijvoorbeeld aan dat na het willekeurig toevoegen van missing values aan een complete dataset, met deze nieuwe definities zo goed als alle oorspronkelijk association rules teruggevonden kunnen worden. Er wordt echter geen aandacht besteed aan een efficiënte implementatie, wat noodzakelijk is vanwege de eigenschappen van deze nieuwe maten.

De nieuwe support, die wel adequaat is, is echter niet meer monotoon. Dit maakt het onmogelijk om hem te gebruiken bij het doorkruisen van de subset lattice, zoals met de oorspronkelijke definitie wel gedaan kon worden. Hiertoe introduceer ik *extensibility*, een nieuwe eigenschap voor itemsets. Existensibility is verwant met het frequent zijn van een itemset (en met zijn representativiteit), maar is monotoon, wat het bruikbaar maakt bij het doorkruisen van de subset lattice. Het *XMiner* algoritme (eXtensible itemset Miner), dat gebaseerd is op het Eclat algoritme wordt gepresenteerd, en aan de hand van experimenten wordt aangetoond dat het effectief en efficiënt werkt, door het te vergelijken met een baseline-algoritme.

Hierdoor hebben we echter slechts nog maar een frequent set mining algoritme, terwijl association rule mining natuurlijk het genereren van regels vereist (hoewel frequent set mining op zich ook al interessant kan zijn). Spijtig genoeg, net zoals support niet meer monotoon is, kunnen regels niet meer zo vanzelfsprekend gegenereerd worden uit de frequente itemsets. Het kan zelfs nodig zijn om de support van infrequente sets te moeten berekenen om de confidence van een aanverwante association rule te kennen. Door XMiner zo aan te passen dat deze itemsets, die wel noodzakelijk zijn naast de echte frequente itemsets, allemaal gevonden worden, (maar zonder de volledige subset lattice te doorkruisen!), kan er gegarandeerd worden dat dit toch vrij goed, albeit niet optimaal verwezenlijkt kan worden. Zodra dit is bereikt, is het genereren van association rules opnieuw een eenvoudige zaak.

Chapter 1

Introduction

Missing values comprise an important and unavoidable problem in Data Mining, and in data management and analysis in general. Conventional mining techniques are not capable of working with missingness directly, and require some form of (likely biased) workaround or preprocessing at the least. In this work I provide an effective theoretical solution, as well as an efficient algorithm, XMiner, for inherently mining association rules in databases with missing values.

In the Association Rule Mining (ARM) model [1], the goal is to find all rules of the form $X \Rightarrow Y$ in a transactional database $\mathcal{D}$, with X and Y sets of items, that intuitively, occur frequently and have a high probability. To assess the goodness of such rules, two measures for association rules are defined, *support* and *confidence*. Typically, association rule mining algorithms can be divided into two parts: frequent set mining and rule generation, *i.e.*, first all itemsets with high support are found, from which all confident rules are then generated. The latter step is rather straightforward. For frequent set mining, the subset lattice over all items spanning the exponentially large search space of itemsets, is traversed using the monotonicity of the support measure. This simple yet fundamental property states that no superset of an infrequent itemset, can itself be frequent. This allows large sublattices with an infrequent set as root to be pruned completely. The traversal can be breadth- or depth-first; many algorithms exists, the most famous ones being Apriori [2], Eclat [14] and FP-Growth [6]. While association rule mining is conventionally defined for the transactional database model, it is easily (though not entirely trivially) extended to and implemented for relational databases. Since these databases are more real-world, and the problem of missing data occurs more naturally there, this work will deal with the relational model instead.

The introduction of missing data in the input database brings many problems for an ARM algorithm. The support and confidence measures become distorted, which could result in the loss of good or fabrication of bad

rules. The actual cause of missingness, the missingness mechanism, must be considered. Are the values missing at random? Can nulls be treated as a separate value? Can useful estimations of the missing values be made? The usual approach is to either impute the missing data *e.g.* by using a mean attribute value, or to remove tuples with nulls. It is clear that these techniques can severely distort the distribution of the data, which can have a bad result on the output.

In this work the support and confidence measures are redefined in a backwards compatible manner (extended if you will), along with a newly defined measure *representativity*, which expresses non-missingness or observation. These definitions derive from previous work by *Ragel and Crémillieux* [8]. The emphasis in their work is on the properties and good applicability of these measures. For example, they show that after randomly inserting nulls in a complete database, most if not all original association rules can still be retrieved. Unfortunately they do not focus on the implementation of their extended measures, which is a necessity due to some of the properties of the measures.

The new support measure, though adequate, no longer exhibits the monotonicity property. This makes it impossible to be used as a subset lattice traversal guide, as was the case with the regular definition of support. Therefore I introduce *extensibility*, a new itemset property. Extensibility is related to the support of an itemset (as well as its representativity), but it is monotone and hence this property will aid traversal of the search space. The *XMiner* algorithm (for eXtensible itemset Miner), based on the Eclat algorithm (and with some elements which are similar to the MaxMiner algorithm), is given and through experimentation it is shown to be effective and efficient, by being compared against a straightforward baseline algorithm.

This leaves us however with only an itemset mining algorithm, whereas association rule mining of course requires the generation of actual rules from them (although itemsets in themselves can also be interesting). Unfortunately, just as the extended support measure was no longer monotone, the extended confidence measure causes trouble as well. The cause of this lies with the straightforwardness of rule generation from frequent itemsets, which it is anything but now. Indeed, it might even be so that the support count of an itemset that is infrequent, is required to compute the confidence of an association rule. By adapting XMiner in such a way that the itemsets, necessary not only for frequent set mining but for rule generation too, are all found (yet without traversing the entire search space!), it is ensured that this can be accomplished fairly well, albeit not optimally. Once this has been achieved, rule generation is again a straightforward matter.

The next chapter reviews basic frequent itemset mining and association rule mining, chapter 3 explains some missingness mechanisms and techniques that are commonly in use to try to deal with missingness. Chapter 4 then gives the definitions and properties of the extended itemset and association

rule measures, and supplies the algorithms that provide efficient implementation. Experimental results are listed in chapter 5, chapter 6 concludes this thesis.

Chapter 2

The Association Rule Model

This chapter briefly reviews Association Rule Mining. The first section discusses Frequent Set Mining. Then, section two covers basic concepts. Section three describes the algorithms Apriori and Eclat. Section four touches on some generalizations.

2.1 Association Rules and Frequent Sets

Association rule mining was introduced by Agrawal et al. [1]. Due to modern technologies, it is possible for organizations to acquire and store lots of data, finding useful and unexpected patterns in this data can be an interesting but complex task. A typical example used is that of a supermarket database, which contains information about customer transactions, such as customer, date, products purchased, price, etc. Analysis of this data can be useful to the store for making marketing decisions such as pricing strategies or product placement in the store. The process of finding useful information in such databases is called market basket analysis.

Staying with our supermarket example, we are interested in finding groups of products (items) that are frequently purchased together, and associations between these products, such as "beer and chips are bought together" or "if a customer buys beer, he also buys chips". As an association rule this would be: "beer" $\Rightarrow$ "chips". Of course we only want the rules that are useful to us, so we define measures for these rules to express their relevance. First of all, if only a few customers bought "beer" and "chips" together, the set of items {"beer", "chips"} will have no statistical significance. So we set a minimum support threshold, to exclude these itemsets. When we know that "beer" and "chips" are frequent, we can construct rules with them. The rule "beer" $\Rightarrow$ "chips" expresses the conditional probability between the itemsets {"beer"} and {"chips"}. The strength of this rule is called the confidence, *e.g.* a confidence of 80% means that $Pr(\text{"}chips\text{"}|\text{"}beer\text{"})$=80% *i.e.* "80% of all customers who bought beer,

also bought chips". Again we can set a threshold, to eliminate those rules that are not confident. Both measures are necessary, in that no one implies the other.

Formally, we are given a database $\mathcal{D}$ of transactions, where each transaction t contains a number of items from a finite item universe $\mathcal{I}$. A set $I \subseteq \mathcal{I}$ is called an itemset, an itemset with cardinality k is called a k-itemset. The frequency (or count) of an itemset I is defined to be the number of transactions t in $\mathcal{D}$ that support I, *i.e.* the transactions that have I as a subset. An association rule is a conditional implication of the form $X \Rightarrow Y$, where X and Y are itemsets for which $X \neq \phi$, $Y \neq \phi$ and $X \cap Y = \phi$.

Definition 1 *The support of a rule $X \Rightarrow Y$ is the frequency of the itemset $X \cup Y$ in the database. Formally,*

$$support(X \Rightarrow Y) = Pr(X \cup Y) = \frac{count(X \cup Y)}{|\mathcal{D}|}$$

Since $|\mathcal{D}|$ is constant, the support can also be defined as an absolute number, in stead of relative.

Definition 2 *The support of a rule $X \Rightarrow Y$ is the conditional strength between the itemsets X and Y. Formally,*

$$confidence(X \Rightarrow Y) = Pr(Y|X) = \frac{count(X \cup Y)}{count(X)}$$

When mining association rules, we want to find all rules above certain (user defined) minimum support and confidence thresholds, from here on these will be referred to as *minsup* and *minconf* respectively. Rules and itemsets that satisfy *minsup* are called frequent, rules that satisfy *minconf* are called confident or strong.

The problem of mining for association rules can be decomposed into two steps: first find all frequent itemsets X, this step is called frequent set mining. In its most basic form (as described above) it is sometimes referred to as one dimensional boolean frequent set mining, since it only deals with the presence of items. (Generalizations are discussed in 2.4.) Afterward, from the frequent sets, generate all confident rules $X \backslash Y \Rightarrow Y$, for $Y \subset X$. The first step is the most computationally intensive. Once this is done, rule generation is straightforward. Therefore algorithms focus on frequent itemset discovery, as will this thesis. However some caution must be taken when missing values are to be taken into account, as will be seen later.

2.2 Basic Concepts

The single most important property used in frequent set mining algorithms, is the monotonicity principle, aka the Apriori property. It simply states that the support of a superset Y of X, must always be equal or less than the support of X. From this we infer that all subsets of a frequent itemset must also be frequent, or alternatively, no superset of an infrequent itemset can be frequent.

$$X \subset Y \Rightarrow \text{supp}(X) \geq \text{supp}(Y)$$

Figure 2.1: The Holy Grail

Let us consider that we are working in a databases with items from a universe $\mathcal{I}$ with n items. Let's look at the subset lattice over $\mathcal{I}$. Figure 2.2 shows an example for $n = 4$.

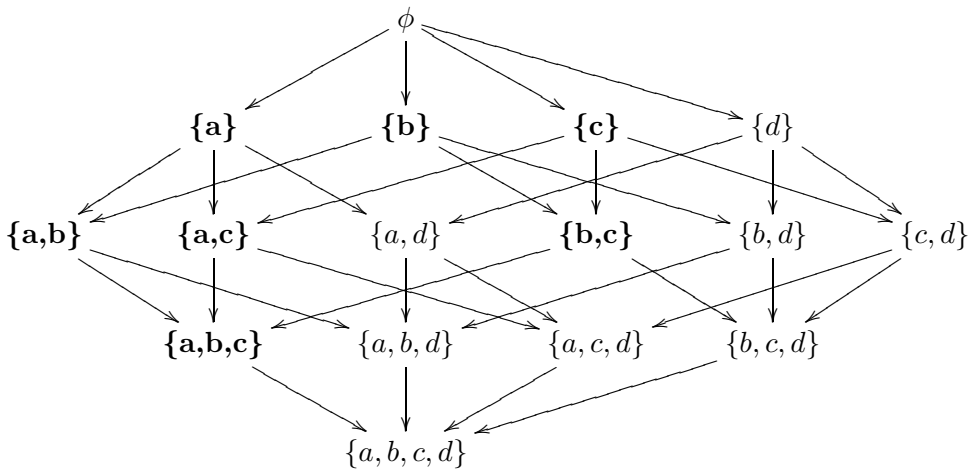


Figure 2.2: Subset lattice over $\mathcal{I} = \{a, b, c, d\}$

When designing an algorithm that mines for frequent sets, we traverse the subset lattice in some way, and we want to

- find all frequent itemsets
- visit as few non-frequent sets as possible.

Since the size of the lattice is exponential in the number of items, the last part is key. Using the monotonicity principle, we can scrap all supersets of an infrequent itemset from consideration, and do the same to an itemset of whose subsets are all found to be frequent. The lattice can be divided in

two parts by a frequency border. As illustrated in figure 2.2, the itemsets in bold are frequent, the other ones are not. Many lattice traversal strategies can be employed, usually a breadth-first (the most famous being Apriori) or depth-first (such as Eclat or FP-Growth) technique is used, although other possibilities exist such as best-first by some heuristic.

Another important aspect to consider is the database layout. When using a horizontal layout, the database is organized as a list of transactions, with a transaction id (*tid*), followed by the items contained in it. When counting the support of an itemset, we must go over the entire database, and for each transaction check whether the itemset is supported by the transaction. This subset checking can contribute to a lot of overhead, and furthermore, for each iteration we must rescan the entire database, which can be very costly, especially if the database cannot in its entirety reside in memory. Several algorithms try to overcome this conundrum. Partition [10] for example, reduces I/O cost by partitioning the database in parts that fit in memory, and then applying Apriori locally to obtain a list of possible frequent sets, it then scans the database a second time, to verify them. A database with a vertical layout consists of entries that are items, followed by a tid-list of the transactions that contain the item. This layout is far superior, since support counting becomes as easy as counting the number of elements in a tid-list. When we want to count the support of X and Y, we simply intersect their tid-lists to obtain the tid-list of $X \cup Y$.

Figures 2.3 and 2.4 show examples of each layout on identical databases.

tid	"beer"	"chips"	"wine"	"cereal"
1	0	1	1	0
2	1	1	0	1
3	1	0	0	0
4	0	0	1	1

Figure 2.3: Horizontal database layout

item	"beer"	"chips"	"wine"	"cereal"
1	0	1	1	0
2	1	1	0	1
3	1	0	0	0
4	0	0	1	1

Figure 2.4: Vertical database layout

2.3 Two Frequent Set Discovery Algorithms

In this section we will describe two of the best known algorithms for discovering boolean frequent itemsets. They are *Apriori* and *Eclat*. Extensive and detailed descriptions can be found in (among others) [3] [13]. Many variations and specialized optimizations for these algorithms exist.

2.3.1 Apriori

The Apriori algorithm was introduced by Agrawal et al. [3]. It employs a breadth-first, generate-and-test strategy on the search space.

It starts with a scan over the database, to determine the set of frequent items (frequent 1-itemsets). Then it subsequently generates frequent itemsets of increasing length. In step k Apriori generates the set C_k of candidate frequent k-itemsets, using the itemsets of length $k-1$ that were found to be frequent in the previous step, and stored in the set L_{k-1}. A candidate is only generated if all of its $(k-1)$-subsets are frequent; otherwise the set itself cannot be frequent. This is the pruning step that uses the monotonicity principle, and it is used to its full power (*i.e.* all subsets are verified). When all candidates of length k are generated, the database is scanned, going over all transactions t, and for each candidate set c in C_k, a counter is increased if t supports c. All candidates that have support larger than *minsup* are put in L_k. The process continues until no new itemsets are discovered. The algorithm is shown in pseudo-code below.

Algorithm 1 Apriori(database $\mathcal{D}$, *minsup*)

1. $L_1 = \{\text{frequent 1-itemsets}\}$
2. **for** $(k = 2;\ L_{k-1} \neq \phi;\ k++)$ **do**
3. $\quad C_k =$ set of new candidates generated from L_{k-1}
4. $\quad$ **for all** transactions t in $\mathcal{D}$ **do**
5. $\quad\quad$ **for all** candidates $c \in C_k$ **do**
6. $\quad\quad\quad$ **if** $(c \subseteq t.items)$ **then**
7. $\quad\quad\quad\quad c.count++$
8. $\quad L_k = \{c \in C_k | c.count \geq minsup\}$
9. return $\bigcup_k L_k$

2.3.2 Eclat

The Eclat algorithm was introduced by Zaki et al. [13]. It employs a depth-first lattice traversal and uses a vertical database layout. Candidates of length k are generated by combining two frequent $(k-1)$-itemsets sharing a common $(k-2)$-prefix. Note that although the monotonicity property is used here, not all subsets of a candidate itemset are checked for infrequency.

However, results have shown that Eclat still has a considerable advantage over Apriori [13]. Optimizations exist such as using diffsets [12] or right-most DFS for complete subset pruning. The algorithm is depicted in pseudo-code below. It is initially called with the set of frequent 1-itemsets, obtained from a single database scan.

Algorithm 2 Eclat(itemset P)

```
1. for all X_i ∈ P do
2.   for all X_j ∈ P with j > i do
3.     X.tids = X_i.tids ∩ X_j.tids
4.     if |X.tids| ≥ minsup then
5.       P_i = P_i ∪ {X}
6.   for all P_i ≠ φ do
7.     Eclat(P_i)
```

2.4 Generalizations

The boolean association rule model, as defined above, can be extended in various aspects.

- One way is to change from a transactional database model to a relational one. In our original definition we were considering transactions, which were just unstructured sets of items either present or not. Relational databases are more complex and realistic, as each tuple has values for a set of attributes, which can be boolean, categorical or continually valued and hence more expressive. (It is also in these databases that the problem of missingness is more naturally occurring.) The transition between transactional and relational databases is simple but not trivial or without complication. A relational database can be constructed from a transactional one by having a boolean attribute for each possible item - creating a sparse database with a gigantic amount of attributes. Inversely, a transactional database can be constructed from a relational one, by constructing the item universe $\mathcal{I}$ for all possible attribute-value combinations $(A = a_i)$ in the relational database, and translating tuples to transactions accordingly. Doing this naively though, frequent set mining would be inefficient if we omitted the fact that no two items $(A = a_i)$ and $(A = a_j)$, for the same attribute A can ever be combined in an itemset (or occur together in a transaction for that matter).

- Another generalization, is to impose a hierarchical structure on the items, a taxonomy with several layers of concepts, *e.g.* the classification "$Food$" > "$Dairy$" > "$Milk$" for products in a warehouse [11]. This

might be useful to mine very broad rules such as "Dairy"⇒"Fruit", whereas too specific rules such as "Skim milk" ⇒ "Pine apple" are not frequent enough to be detected. This generalization itself can be expanded as well by also allowing cross-level rules, or maintaining different support thresholds at different concept levels.

- Sometimes it is not necessary to mine all frequent sets, but we are only interested in *maximal* frequent itemsets. A frequent itemset I is called maximal if none of its supersets is frequent. For example, the rule "Bread" ⇒ "Milk" "Eggs" contains the rule "Bread" ⇒ "Milk" completely, and hence the latter is superfluous. From the set of all maximal frequent itemsets we can infer all frequent sets, by generating their subsets (however we can not derive their supports). Note that the maximal frequent itemsets alone completely describe the frequency border in the subset lattice. It has been shown that by using pruning techniques for this specific situation, maximal set mining can be accomplished efficiently, *i.e.* linearly in the number of maximal frequent sets, whereas the number of all frequent sets can be exponential in the length of the longest (maximal) itemset [4].

Chapter 3

Missingness

This chapter will give an overview of different missingness mechanisms, and common techniques for dealing with the missing value problem.

Missing data occurs in many real-world datasets. Because of it the complexity of analysis increases, and the results can be influenced considerably. Ideally, we would want to make the same inferences about the data, as if we had the complete dataset (if this is possible at all). A key component here is the *mechanism* at which this missingness occurs, *i.e.* the probability of a value being missing, depending either on other attributes, or its own value. Hence, when analyzing a dataset with missing values, we must take the missingness mechanism into account. Unfortunately, this mechanism is not always known, and assumptions about the data must be made.

3.1 Classification

Missingness can be classified into three missingness mechanisms. They were identified by Rubin and are commonly used among statisticians [9]. For the sake of clarity, we will be working in a relational database. Figure 3.1 gives a very simple example database with no missing values, which will serve as a reference. We will use the following notation in this chapter: A_m denotes that a value for the attribute A is missing, and A_o means a value is observed for the attribute A.

Age	Gender	Income
30	female	40000
37	male	45000
19	male	50000
26	female	55000

Figure 3.1: Database without missing values

3.1.1 MCAR - Missing completely at random

A value is missing completely at random if it is independent of its own value, or the value of any other attribute. The probability that a value of some attribute is missing is always the same. This is random missingness in the intuitive sense of the word. In figure 3.2, with missing values for the attribute "Income", we have $Pr(\text{Income}_m) = Pr(\text{Income}_m|\text{Gender=male}) = Pr(\text{Income}_m|\text{Gender=female}) = Pr(\text{Income}_m|\text{Income<50k}) = \ldots$

Age	Gender	Income
30	female	?
37	male	45000
19	male	?
26	female	55000

Figure 3.2: Database with missing values, MCAR

3.1.2 MAR - Missing at random

A value is (quite confusingly) called missing at random, if it is dependent of one or more other attributes (i.e. their values in the same tuple). For instance when in a survey a question starts with *"In case you answered yes, ... "* and the answer was "no", then the latter question was not applicable, and we have a null value. In figure 3.3 "Income_m" depends on "Gender", and $Pr(\text{Income}_m|\text{Gender=male}) \neq Pr(\text{Income}_m|\text{Gender=female})$.

Age	Gender	Income
30	female	?
37	male	45000
19	male	50000
26	female	?

Figure 3.3: Database with missing values, MAR

3.1.3 MNAR - Missing not at random

A value is called missing not at random, if it is dependent of itself. A typical example is a measuring device incapable of detecting extreme values above a certain threshold, and so produces null values. This is the most difficult mechanism to deal with, since it can not be derived from the data. In figure 3.4 "Income" is missing if "Income<50k", $Pr(\text{Income}_m|\text{Income<50k}) \neq Pr(\text{Income}_m|\text{Income>50k})$.

Age	Gender	Income
30	female	40000
37	male	45000
19	male	?
26	female	?

Figure 3.4: Database with missing values, MNAR

Please note that the previous examples are specially crafted for the purpose of illustration, and perhaps a bit too simple. The statements made about equality of probabilities must be interpreted as probabilities of missingness, not as support, when trying to infer whether data is *e.g.* MAR, supports will not be exactly equal.

Furthermore, note that the MNAR mechanism is not very important in the association rule setting. Suppose we have two attributes A and B with missing values for B, MNAR. If in the 'complete' database there are no rules $(A = a_i) \Rightarrow (B = b_j)$ or $(B = b_j) \Rightarrow (A = a_i)$, then the missingness for values of B has no influence on the results. If such rules are present, then in the incomplete database these rules become $(A = a_i) \Rightarrow (B =?)$ or $(B =?) \Rightarrow (A = a_i)$. So the mechanism is in fact MAR. A possible side effect could be that many different values b_j for B are missing, and falsely a MAR $(A = a_i) \Rightarrow (B =?)$ rule is discovered. However, this rule can be interpreted as an actual MAR rule, for the aggregate values of B that are MNAR. For simplicity the missingness mechanism will be assumed to be either MCAR or MAR from here on.

3.2 Dealing With It

Conventional methods to treat missing data - not only in data mining - can be divided into two parts. These methods either remove or add information to the database. Both approaches try to obtain a complete data set on which existing algorithms can be applied. Of course this can heavily bias the results and estimators such as mean or variance.

3.2.1 Removing data

The first and simplest solution is to delete all tuples that have nulls. Though this does indeed create a complete data set, it is obvious that many information is discarded that might have been useful. In fact, if for example we have a database with 20 attributes and 1000 tuples, and a probability of 5% for a value of any attribute being missing (MCAR), then the probability that a tuple has no nulls at all is only 35%. Moreover, if the data is not MCAR,

then the results will become skewed, and estimators such as mean will become biased, since more tuples with a certain value for a certain attribute will be removed. A similar deletion technique is removing all attributes that have missing data, but this is even more ridiculous.

3.2.2 Adding data

The second alternative to obtain a complete database is to fill in the gaps. Several techniques exist, some of the most common are listed below.

Mean and mode imputation

With mean imputation, all nulls for a given attribute are replaced by the mean of all observed values for that attribute. While for attributes with very few missing values that are MCAR this might not be too bad, it is clear that when many values are missing, or they are not MCAR, this severely deforms the data. For one, standard deviation will be underestimated. Of course this technique is only applicable to continuously valued attributes, not categorical. Mode imputation is more (only) applicable to categorical attributes, it replaces the nulls for an attribute with the most observed value for that attribute. Similar remarks as with mean imputation apply here, especially variance will be lowered.

The missingness category

Another possibility is to add an extra category that represents the missingness for an attribute. Clearly if the missingness mechanism is not MAR, (and even then), it is possible that very dissimilar groups of attributes become grouped under this new category.

Multiple imputation

Rather than filling up the dataset once to obtain a complete dataset, another possibility consists of creating several complete datasets, each with different values for the missingness imputed. Each of these datasets can then be analyzed separately, after which the means and variance of all of these results are used to obtain the final results. Although multiple imputation can yield results that are far better than the previous methods, it is clear that it is computationally more expensive, and that any derived inferences from this technique are still only valid within some acceptability interval.

Chapter 4

Association Rule Mining with Missing Values

This chapter describes the problem of missing values for association rule mining, extends and defines some new itemset and association rule measures, along with some theoretical properties, and implements them in an algorithm with several possible optimizations.

4.1 The Problem

Let us first look at an example, and see why the conventional itemset and association rule measures are insufficient. We have two simple databases with 4 tuples and 3 attributes. The second one is simply the first one, but with some missing values.

t	A	B	C
1	a_1	b_2	c_1
2	a_1	b_2	c_2
3	a_2	b_2	c_2
4	a_1	b_1	c_1

t	A	B	C
1	a_1	b_2	c_1
2	**?**	b_2	c_2
3	a_2	b_2	**?**
4	a_1	**?**	c_1

Figure 4.1: Identical databases, without and with missing values

When using the classic definitions of support and confidence, the rule $(A = a_1) \Rightarrow (B = b_2)$ has a support of $\frac{2}{4} = 50\%$, and confidence $\frac{2}{3} = 66\%$ in DB1. The rule $(A = a_1) \Rightarrow (C = c_1)$ has exactly the same support and confidence. In DB2, due to the missing values, the support of rule 1 has dropped to $\frac{1}{4} = 25\%$, while that of rule 2 has remained the same. The confidence of rule 1 has also dropped, to $\frac{1}{2} = 50\%$, while the confidence of rule 2 has become $\frac{2}{2} = 100\%$. In general support can drop, possibly below *minsup*, and confidence can either drop, possibly under *minconf*, or

rise, making a rule over-confident. This is because we no longer take into account the transactions with missing values, while it might be possible that the ? hides a value that is useful. The reason for this distortion is that we recognize the missing values as some value that is never equal to the one we are counting. We completely ignore the special meaning of the ?'s - they may or may not hide a value of interest to us, we just don't know.

4.2 Definitions

To solve this problem, some new and extended itemset and association rule measures need to be introduced. They take into account the possibility of missing values, and treat them accordingly. These quite straightforward measures were defined by Ragel and Crémillieux [8], who in their work applied them to find rules in a database with missing values inserted at random. By inspecting the results, it was shown that most, if not all of the original rules could be retrieved, and almost no new (false) rules were introduced. However, no mention of algorithmic implementation is made, except that a version of Apriori was used. Also, the third measure (representativity, which will prove to be key), never even seems to be used after being defined. I assume that a traditional frequent set mining algorithm was used, and post-processing was applied on the results. Here I will formally (re)define the measures, and explain them.

First, let us lay down some notation. We are working in a relational database $\mathcal{D}$ (which as seen earlier, can be viewed as a transactional database). Each tuple has a transaction/tuple identifier *tid*. Attributes $A, B, C, \ldots$ and their values $a, b, c, \ldots$ will be written in uppercase and lowercase respectively. An item is an attribute value pair, written as $(A = a_i)$. The *missingness item* is written as $(A =?)$, indicating a missing value for the attribute A. (Note that although this item's notation is similar to that of a proper item, mindlessly treating it as such is wrong.) Also, we have the *attribute item* $(A = *)$, which is used when a value (any value) is observed for the attribute A. (The same remark as for the missingness item applies).

Itemsets are sets of these pairs, in general they can also contain missingness and attribute items. This generality allows us to mine interesting rules that carry information about the missingness mechanisms present in the database. However, this extension is not inherently more difficult, so I will not focus on it here. The only tricky part lies with the double role that attribute items play (due to the upcoming definitions), but through careful implementation this is quite manageable. Furthermore, we will assume that itemsets are consistent, in that no two items with the same attribute (*e.g.* $(A = a_1)$ and $(A = a_2)$, or $(B = b_1)$ and $(B =?)$) can be in an itemset together. The only trivial exception to this is *e.g.* $(C = c_1)$ and $(C = *)$, but since the latter is implied by the former, it can be omitted.

By the *count* of an itemset, we mean the absolute number of transactions in $\mathcal{D}$ that support that itemset:

Definition 3 $count(X) := |\{t \in \mathcal{D} | X \subseteq t.items\}|$

Finally, the *attribute set* of an itemset X is defined as:

Definition 4 $X^* := \{(A = *) | (A = a_i) \in X\}$

For example, $\{(A = a_1), (B = b_2), (C = *), (D = ?)\}^* = \{(A = *), (B = *)\}$.

4.2.1 Support

When looking at what went wrong in example 4.1, we see that we counted all tuples that support X, and divided it by the total number of tuples $|\mathcal{D}|$. However, some of those tuples have missing values for some attributes of X. Since the ?'s may or may not hide complete transactions that support X, we should omit these tuples. Only the tuples that have no missing values for any of the attributes in X^* will be considered - although those tuples might still have missing values for other attributes! Statistically this is expressed by $Pr(X|X^*)$. We can now define the new support measure:

Definition 5

$$supp(X) := \frac{count(X)}{count(X^*)}$$

4.2.2 Confidence

A similar observation can be made for the confidence of an association rule. For the rule $X \Rightarrow Y$ there might be missing values for the antecedent Y. The solution here is to again only consider those tuples that do not have missing values for Y. Statistically this is expressed by $Pr(Y|X \cup Y^*)$.

Definition 6

$$conf(X \Rightarrow Y) := \frac{count(X \cup Y)}{count(X \cup Y^*)}$$

4.2.3 Representativity

Apart from confidence and support, representativity is introduced as a new measure. The rationale behind it is to limit the influence of itemsets that are scarcely observed (*i.e.* many tuples in $\mathcal{D}$ have missing values for some of the attributes of the itemset), on confidence and support. In other words the sample of $\mathcal{D}$ that has no missing values for the attribute set of X must be a representative sample.

Definition 7

$$rep(X) := \frac{count(X^*)}{|\mathcal{D}|}$$

4.2.4 Extensibility

One more definition is due, I define the *extensibility* property of an itemset (which is not a measure). Later we will see why this is much needed.

Definition 8 *An itemset X is called extensible, if an itemset Y exists such that $X \cup Y$ is frequent and representative,* i.e.

$$\exists Y : \begin{cases} supp(X \cup Y) \geq minsup \\ rep(X \cup Y) \geq minrep \end{cases}$$

Note that Y may be the empty set, so all itemsets that are frequent and representative are trivially extensible. Furthermore, it goes without saying that extensibility is monotone.

4.2.5 Remarks

- Because of the new definition of confidence, we can no longer easily construct the confident rules from the frequent sets. Before, the confidence of a rule $X \Rightarrow Y$ was equal to $\text{supp}(X \cup Y)/\text{supp}(X)$. This is not true anymore, since

  $$\frac{\text{supp}(X \cup Y)}{\text{supp}(X)} = \frac{\text{count}(X \cup Y)/\text{count}(X^* \cup Y^*)}{\text{count}(X)/\text{count}(X^*)}.$$

  Statistically $Pr(Y|X \cup Y^*)$ only equals $Pr(Y|X)$ if X and Y are independent.

- As was the case with the classic support definition, representativity is defined relative to $|\mathcal{D}|$. In the following however representativity can be either relative or absolute depending on the context.

- Looking closely at the definition of representativity, it is very reminiscent of another definition: the conventional definition of support. In a way, representativity can be considered to be support on the level of missingness, or rather, observation. The following equality illustrates this (since for any X, $(X^*)^* = \phi$):

  $$\text{rep}(X) = \frac{\text{count}(X^*)}{\text{count}((X^*)^*)}.$$

  This implies a hierarchy among different kinds of items or itemsets, as shown in figure 4.2. A single attribute A has two children $(A = *)$ and $(A =?)$. The former still has more specific children, namely the attribute-value pairs $(A = a_i)$. Implementing the new definitions in a generalized itemset mining algorithm (which has already been studied extensively [11]) is not possible, since the generalized monotonicity principle does not hold, *i.e.* the support of more general itemsets is not necessarily higher than the support of more specific itemsets.

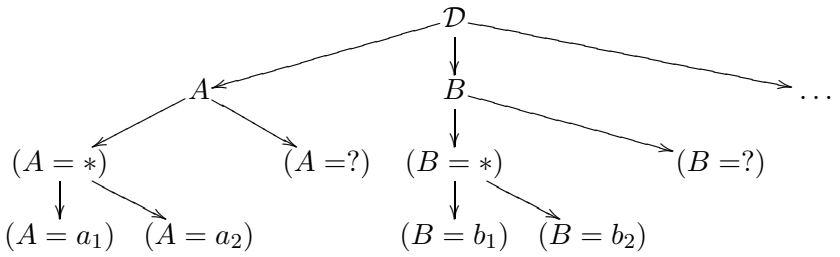


Figure 4.2: Hierarchy of items

- Note that all of the previous definitions are backwards compatible with the classic definitions, where missingness is absent. In that case, representativity is rather trivial, always being 100%, and extensibility is the same as frequency.

4.3 Properties

The new measures now correctly take missingness into account, and when there is no missingness, they revert back to the old measures. Unfortunately the most important property is now gone - the support measure is no longer monotone. In $\text{count}(X)/\text{count}(X^*)$ both nominator and denominator themselves show monotone behavior, but the new support does not. In fact $X \subset Y \not\Rightarrow \text{supp}(X) \leq \text{supp}(Y)$. In the subset lattice, frequent and infrequent sets are no longer separated by a simple border. In stead, it is quite chaotic and doesn't even have to be linearly connected. As a result traversing the lattice is no longer a matter of exploring the search space toward the border, using the monotonicity of support as a guide. When an infrequent set is encountered we cannot stop, there still might be isolated 'enclaves' of frequent itemsets further on. This is illustrated in figure 4.3.

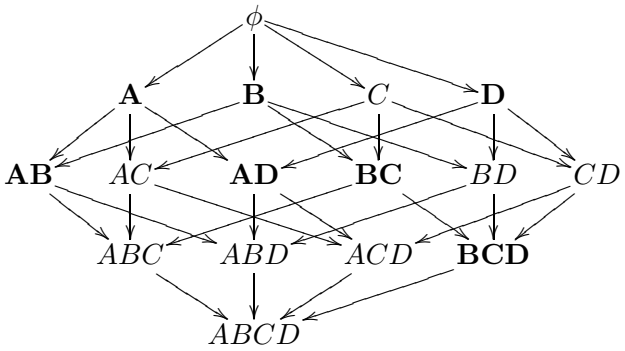


Figure 4.3: Subset lattice with bold frequent sets

In stead of support, we need a different pruning test, preferably monotone and easy to compute, yet somehow related to support. As the attentive reader may have guessed by now, this will be the extensibility property.

Theorem 1 *Let S_X be the set $\{Z \supseteq X | Z$ representative and frequent$\}$ for an itemset X. We define the function m as $m(X) := \min\{rep(Z) | Z \in S_X\}$, or $m(X) := +\infty$, if $S_X = \phi$. We then have*

$$X \text{ is extensible} \Leftrightarrow \frac{count(X)}{m(X)} \geq minsup$$

Proof. If X is extensible, $S_X \neq \phi$, so $\text{count}(X)/m(X) \geq \text{supp}(X) \geq minsup$, since $minrep \leq m(X) \leq \text{rep}(X)$. If X is inextensible, $S_X = \phi$, so $\text{count}(X)/m(X) = 0 < minsup$.

We now have a computational way of ascertaining the extensibility of an itemset, so we can use this to traverse the subset lattice, and prune all infrequent and/or unrepresentative itemsets. At this point it must be noted that extensibility does not imply frequency. It may be possible that using this theorem, infrequent 'enclaves' among frequent itemsets in the subset lattice, are (correctly of course) labeled as extensible and involved in computation. However, most if not all algorithms, either breadth-first or depth-first, incrementally use itemset information from previous steps (for example in *tid*-list construction), and so these itemsets are unavoidable anyway. Without them it would be impossible to compute the count of their supersets, some of which are indeed frequent (as told by the theorem).

Unfortunately we have not really solved the problem at all, since the aforementioned theorem is rather paradoxical. In order to prune a branch in a subset lattice *i.e.* the supersets, we must first compute all of them to obtain the function value $m(X)$. Hence, theorem 1 cannot be directly applied as such in an algorithm. We can however use a weaker form, described in this corollary.

Corollary 1 *For an itemset X, let $k \leq m(X)$. Then*

$$\frac{count(X)}{k} < minsup \Rightarrow X \text{ is not extensible.}$$

Proof. Trivial verification.

4.4 Frequent Set Algorithms

This section describes a baseline algorithm and the XMiner algorithm for computing all frequent sets from databases with missing values. Due to the extended definition of confidence however, these are not sufficient for also computing all itemsets necessary to generate all confident association rules. This matter will be deferred to the next section.

4.4.1 Baseline algorithm

First, a straightforward solution is proposed which will serve as a baseline for comparison. The lower bound of $m(X)$ is for all itemsets fixed to *minrep* (see Corollary 1), since $m(X) \geq minrep$ is always true. For an itemset X, first $\text{rep}(X) \geq minrep$ is checked and if true $\text{count}(X)$ counted. If $\text{count}(X)/minrep < minsup$ the algorithm concludes X is inextensible. Otherwise the algorithm must continue with the supersets of X.

The fact that the $m(X)$ lower bound is fixed for all itemsets simplifies implementation. I have chosen to adapt Eclat [14] for its simplicity and speed. It makes a depth-first traversal, for each itemset maintaining a *tid*-list of transactions supporting that itemset. At each step itemsets with a common prefix are combined to obtain larger ones (here the monotonicity of extensibility is used). The implementation also uses diffsets as an optimization [15].

The baseline algorithm is equivalent to outputting all representative itemsets whose count is larger than $minsup \cdot minrep$, and filtering out the frequent ones in a post-processing step. For the sake of brevity the itemsets in the pseudo code below are denoted by their last item, hence omitting the common prefix.

Algorithm 3 Baseline(set of itemsets P)

Require: P is a set of representative and possibly extensible ordered itemsets with a common (omitted) prefix

```
1.  for all X_i ∈ P do
2.    Representation(X_i, P)
3.    for all values x_i of X_i do
4.      for all X_j ∈ P with j > i do
5.        if count(X_ij = *) ≥ minrep then
6.          for all values x_j of X_j do
7.            (X_ij = x_ij).tids = (X_i = x_i).tids ∩ (X_j = x_j).tids
8.            if count(X_ij = x_ij)/minrep ≥ minsup then
9.              P^i = P^i ∪ {(X_ij = x_ij)}
10.           if count(X_ij = x_ij)/count(X_ij = *) ≥ minsup then
11.             report (X_ij = x_ij) as frequent
12.     Baseline(P^i)
```

Algorithm 4 Representation(X_i, P)

```
1. for all X_j ∈ P with j > i do
2.   compute (X_ij = *).tids = (X_i = *).tids ∩ (X_j = *).tids
```

4.4.2 XMiner

I will now present the XMiner algorithm, by building it up incrementally to improve it. For this, $m(X)$ will be approximated by a non-constant lower bound, *i.e.* it is dependent of X. To reiterate, $m(X)$ is defined as $\min\{\text{rep}(Z)|Z \in S_X\}$, where $S_X = \{Z \supseteq X|Z \text{ frequent and representative}\}$. Obviously we will not generate and test all supersets of X, when all we need is a representativity. We therefore approximate S_X by its superset $S'_X = \{Z \supseteq X|\text{rep}(Z) \geq \mathit{minrep}\}$, which might include non-frequent sets. We now no longer need to count the occurrences of Z's, only their attribute sets Z^*, an exponential reduction in number (for each attribute item in an itemset there might be several values). This simplified approach is equivalent to computing $\min\{\text{count}(Z^*) \geq \mathit{minrep}|Z^* \supseteq X^*\}$, so we are searching for the maximal representative superset of X^* (*i.e.* its representativity), hence we adopt a maximal itemset mining strategy, that works with representativity instead of support (we can do this since representativity by itself is monotone). This can be accomplished by making several intersections of *tid*-lists at once in stead of working level-by-level, and thus quickly looking ahead in the subset lattice, to the itemset at the the so-called tail, the longest path below the current itemset in the traversal tree (see figure 4.4). It can be shown that this itemset has minimal representativity among all supersets of the current itemset.[1] If the tail itemset is not representative we replace it with a better bound, *minrep* (as in the baseline algorithm).

Two additional improvements can be made. The first is the ordering of the itemsets. In regular depth-first itemset mining it is often beneficial to order the itemsets by their support, putting the less frequent ones at the roots of larger subtrees, which yields better pruning. (This is not an absolute order on the items, but can be reevaluated at each recursive step of the algorithm.) XMiner orders the itemsets by their representativity instead, and not support because of the erratic behavior of the support measure, and the impossibility of ordering attributes with several possible values (and hence different counts). This not only improves pruning based on *minrep*, but also returns a better approximation for $m(X)$. The second improvement lies in the computation of the tail itemset. When, while iteratively computing $\bigcap_i X_i^*$, we find that at step j we have $\text{rep}(\bigcap_{i<j} X_i^*) \leq \mathit{minrep}$, we can stop. In conjunction with ordering, this improvement becomes optimal.

In stead of intersecting itemsets to obtain the representativity of the tail itemset, it is also possible to approximate it using basic set theory, with some simple subtractions (similar to the MaxMiner algorithm [4]). Although this approximation is much faster, my experiments have shown this lower bound is far too weak to approximate $m(X)$, such that the overall effect is negative on both the efficiency (number candidate itemsets versus frequent

[1] Actually it is minimal among the supersets in the local subtree below X, see the subsection about local subtree pruning

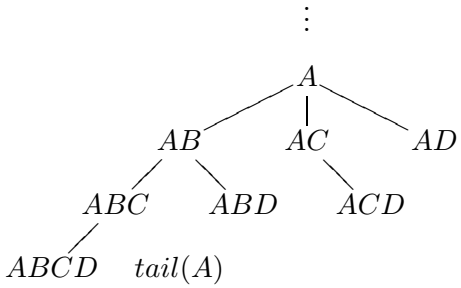


Figure 4.4: Partial subtree with tail of A marked

and representative itemsets) and execution time of XMiner.

Computing the representativities of itemsets and their tails happens inplace with the counting of the itemsets themselves, instead of first collecting all representativity information. As a consequence, we actually aren't computing the tail $\bigcap_i X_i$ of an itemset over all items in the item space, but a reduced set, where each X_i was previously not found to be inextensible or is even frequent. This means that we are - implicitly - considering a set that lies between S_X and S'_X, which again gives a better resulting approximation for $m(X)$.

As a last remark, I would like to point out that XMiner is equally efficient as the Eclat algorithm when no missing values are present in the input database *i.e.* the number of candidate extensible/frequent itemsets is the same. Through the use of diffsets in the implementation, the execution time is also only slightly higher. (At depth greater than one in the subset lattice, all diffsets for representation become empty, resulting in marginalized overhead.) Also, when only a part of the attributes have no missing values, adding items with these attributes to itemsets will be equally efficient.

Local Subtree Pruning

Since the traversal of the search space employed by XMiner happens in a tree rather than a complete lattice, in general pruning consist of pruning only the supersets of an itemset X that have X as a prefix. As a result the approximation of $m(X)$ will be higher. If then $\text{count}(X)/m'(X) < minsup$, we can prune only the local subtree that has X as root. Although a higher $m'(X)$ results in better pruning, we can no longer extrapolate the local inextensibility of X to occurrences of supersets of X elsewhere in the traversal tree, because with those supersets included, $\text{count}(X)/m(X) \geq minsup$ might be true after all. This unfortunately prevents us from maximizing the use of the monotonicity of extensibility (*e.g.* by maintaining a trie of extensible itemsets while doing a right-most depth-first traversal).

Algorithm 5 XMiner(set of itemsets P)

Require: P is a set of representative and possibly extensible ordered itemsets with a common (omitted) prefix

1. **for all** $X_i \in P$ **do**
2. Representation(X_i, P)
3. **for all** values x_i of X_i with count$(X_i = x_i)/m'(X_i) \geq minsup$ **do**
4. **for all** $X_j \in P$ with $X_j > X_i$ **do**
5. **if** count$(X_{ij} = *) \geq minrep$ **then**
6. **for all** values x_j of X_j **do**
7. $(X_{ij} = x_{ij}).tids = (X_i = x_i).tids \cap (X_j = x_j).tids$
8. **if** count$(X_{ij} = x_{ij})/m'(X_i) \geq minsup$ **then**
9. $P^i = P^i \cup \{(X_{ij} = x_{ij})\}$
10. **if** count$(X_{ij} = x_{ij})/$count$(X_{ij} = *) \geq minsup$ **then**
11. report $(X_{ij} = x_{ij})$ as frequent
12. XMiner(P^i)

Algorithm 6 Representation(X_i, P)

1. $(X = *).tids = (X_i = *).tids$
2. $m'(X_i) =$ count$(X = *)$
3. **for all** $X_j \in P$ with $X_j > X_i$ **do**
4. compute $(X_{ij} = *).tids = (X_i = *).tids \cap (X_j = *).tids$
5. **if** $m'(X_i) > minrep$ **then**
6. $(X = *).tids \cap = (X_{ij} = *).tids$
7. $m'(X_i) = \max($count$(X = *), minrep)$

4.5 Rule Generation

As mentioned before, XMiner is still only a frequent set mining algorithm, and does not generate association rules. For conventional association rule mining, rule generation from frequent sets was almost trivial, this is not the case anymore. To reiterate, in their new form support and confidence of an association rule $X \Rightarrow Y$ are defined as count$(X \cup Y)/$count$(X^* \cup Y^*)$ and count$(X \cup Y)/$count$(X \cup Y^*)$ respectively. It is clear to see that

$$\frac{\text{supp}(X \cup Y)}{\text{supp}(X)} = \frac{\text{count}(X \cup Y)/\text{count}(X^* \cup Y^*)}{\text{count}(X)/\text{count}(X^*)} \neq \frac{\text{count}(X \cup Y)}{\text{count}(X \cup Y^*)}.$$

This forms a serious problem, since first of all we must now also mine the more general itemsets of the form $X \cup Y^*$, *i.e.*, that contain attribute items for the antecedent of a possible rule, and secondly it turns out that mining only the frequent of those sets isn't even sufficient. Since we have

$$\text{count}(X \cup Y) \leq \text{count}(X \cup Y^*) \text{ and } \text{count}(X^* \cup Y^*) \leq \text{count}(X^*)$$

we cannot derive any relationship between $\text{supp}(X \cup Y)$ and $\text{supp}(X \cup Y^*)$, due to the lack of monotonicity. So we must choose an order in which we evaluate these sets. If we first verify that $X \cup Y$ is infrequent, we won't have to look at $X \cup Y^*$ since no rules can be formed. Conversely we could first count $X^* \cup Y^*$ and then $X \cup Y^* \ldots$, systematically replacing the attribute items with values. The former approach clearly makes more sense, after all for the latter approach it is possible that after replacing all attribute items in an itemset $X^* \cup Y^*$ there is no frequent set $X \cup Y$ making all previous work useless. Subsequently, experimentation has shown this intuition to be correct.

Now, if $X \cup Y$ is frequent we do not actually have to replace all possible items with attribute items. We don't even really need the support of the sets $X \cup Y^*$, just their count, and count is monotone with respect to generalization of itemsets. This way we can avoid evaluating generalized itemsets (that are partitions of 'normal' itemsets), that won't form confident rules. This is the basic idea behind the rule generating version of XMiner. For example, if an itemset {(A=a),(B=b),(C=c)} turns out to be frequent and representative, we remember its count. Then we systematically proceed with {(A=a),(B=b),(C=*)} etc. If we then find that count({(A=a),(B=b),(C=*)})$\cdot$*minconf* > count({(A=a),(B=b),(C=c)}) (*ie* the rule $(A = a), (B = b) \Rightarrow (C = c)$ is not confident) we don't have to do $\{(A = a), (B = *), (C = *)\}$ anymore since its count will be even larger, and $(A = a) \Rightarrow (B = b), (C = c)$ won't be confident either.

The pseudo-code for XMiner, extended with rule generation is depicted below.

Algorithm 7 XMiner(set of itemsets P)

Require: P is a set of representative and possibly extensible ordered itemsets with a common (omitted) prefix

1. **for all** $X_i \in P$ **do**
2. Representation(X_i, P)
3. **for all** values x_i, * of X_i with count$(X_i = x_i)/m'(X_i) \geq minsup$ **do**
4. **for all** $X_j \in P$ with $X_j > X_i$ **do**
5. **if** count$(X_{ij} = *) \geq minrep$ **then**
6. **for all** values x_j of X_j **do**
7. $(X_{ij} = x_{ij}).tids = (X_i = x_i).tids \cap (X_j = x_j).tids$
8. **if** count$(X_{ij} = x_{ij})/m'(X_i) \geq minsup$ **then**
9. $P^i = P^i \cup \{(X_{ij} = x_{ij})\}$
10. **if** count$(X_{ij} = x_{ij})/$count$(X_{ij} = *) \geq minsup$ **then**
11. report $(X_{ij} = x_{ij})$ as frequent
12. **if** $(X_{ij} = x_{ij})$ has no attribute items **then**
13. store count$(X_{ij} = x_{ij})$
14. **for** the attribute item $(X_j = *)$ **do**
15. **if** $P^i \neq \phi$ **then**
16. $(X_{ij} = x_i).tids = (X_i = x_i).tids \cap (X_j = *).tids$
17. **if** $\exists (X_{ij} = x_{ij})$ without attribute items **then**
18. **if** count$(X_{ij} = x_{ij}) \geq minconf \cdot$ count$(X_{ij} = x_i)$ **then**
19. report corresponding rule if not trivial
20. **if** count$(X_{ij} = x_i)/m'(X_i) \geq minsup$ **then**
21. $P^i = P^i \cup \{(X_{ij} = x_i)\}$
22. XMiner(P^i)

Algorithm 8 Representation(X_i, P)

1. $(X = *).tids = (X_i = *).tids$
2. $m'(X_i) =$ count$(X = *)$
3. **for all** $X_j \in P$ with $X_j > X_i$ **do**
4. compute $(X_{ij} = *).tids = (X_i = *).tids \cap (X_j = *).tids$
5. **if** $m'(X_i) > minrep$ **then**
6. $(X = *).tids \cap = (X_{ij} = *).tids$
7. $m'(X_i) = \max($count$(X = *), minrep)$

Chapter 5

Experimental Results

The XMiner algorithm and the baseline algorithm were tested and compared against each other, on three different databases. The first one is the Sick data set about Thyroid Disease[1]. This database contains 2800 tuples, has 24 attributes and contains some missing values. The second dataset are the results of the Eurovision song contest from 1957 to 2005[2]. In this European contest a limited number of countries can enter and give points to the others, the countries that perform worst cannot enter in the next edition. Over the years more countries could join, recently a semi-final was introduced and countries that do not enter can now cast votes, which was previously not possible. This makes that the database has a lot of missing values. The form is as follows: each tuple represents a year-country combination, and has 2*43 attributes for points given to and received from other countries. The third database is the Census Income dataset[3]. This is a large database, about 100MB in size. It has 33 nominal attributes (after 7 continuous attributes were removed) and nearly 200k tuples. Some attributes have no missing values, other attribute values are missing in up to 50% of the tuples. To investigate the influence of missingness on XMiner, I incrementally created several versions of the database, each with more values of attributes turned into nulls.

I implemented both XMiner and the baseline algorithm in C++ (using diffsets to optimize for speed [15]), compiled them with gcc 3.4.6 (all with optimization -O3), on a machine with an AMD Opteron 64 bit CPU at 2.2 GHz, with 2GB of memory, running Gentoo Linux.

[1] Available from the UCI Machine Learning Repository [5]
[2] Collected and preprocessed by Joost Kok
[3] Available from the UCI KDD Archive [7]

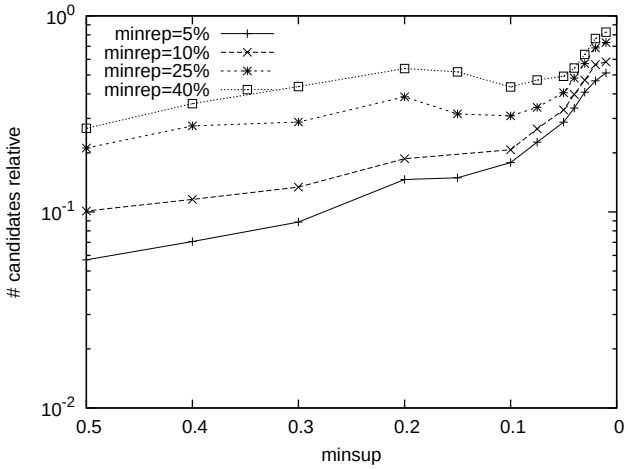
5.1 Sick Dataset

On the Sick dataset the minimum representativity was fixed at 5%, 10%, 25% and 40%, and the minsup was varied each time, to look at the number of candidate itemsets and the execution time of XMiner, relative to the baseline algorithm. From figure 5.1(a) it is clear that XMiner always outperforms the baseline, while 5.1(b) shows the actual number of candidate itemsets of the baseline and XMiner, versus the number of resulting frequent (and representative) itemsets that both algorithms have as output. It shows that XMiner follows quite closely the number of frequent itemsets, while the baseline generates a lot more candidates than necessary, and its course is rather independent of the number of frequent itemsets. Figures 5.1(c) and 5.1(d) show the execution time of the same experiments. They indicate that the execution time corresponds to the number of generated itemsets. The fact that the Sick dataset has some attributes without any missing values, results in a good performance of XMiner, while the baseline algorithm cannot even take advantage of this.

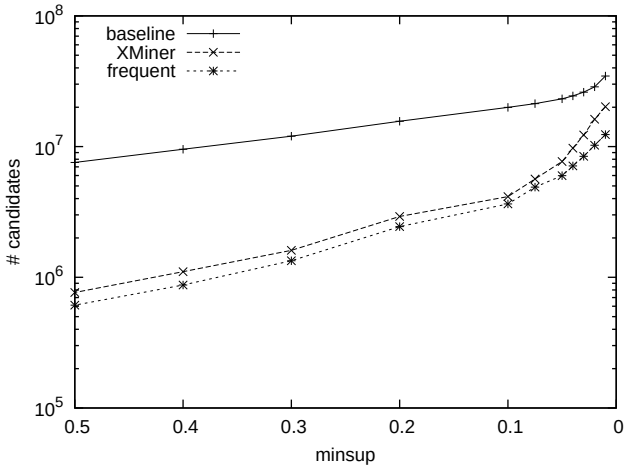
5.2 Eurosong Dataset

Similar experiments as with the sick database were performed on the Eurosong dataset, with several fixed minreps 5%, 10%, 20% and 30%. For the baseline algorithm with parameters minrep=5% and minsup=10%, the process was killed after 4000 seconds, so no comparison with XMiner could be made, and this point is not plotted (figures 5.2(a) and 5.2(c)). This time XMiner's course does not follow the number of representative and frequent itemsets that closely, but still outperforms the baseline algorithm by almost an order of magnitude in generated itemsets (figure 5.2(b)), and a factor of eight in execution time (figure 5.2(d)). The eurosong dataset has more missing values than the sick database (in fact for each attribute there is at least one tuple with a missing value for that attribute). This is why the graphs of the eurosong dataset do not look as spectacular as the sick dataset, but XMiner obviously still outperforms the baseline algorithm.

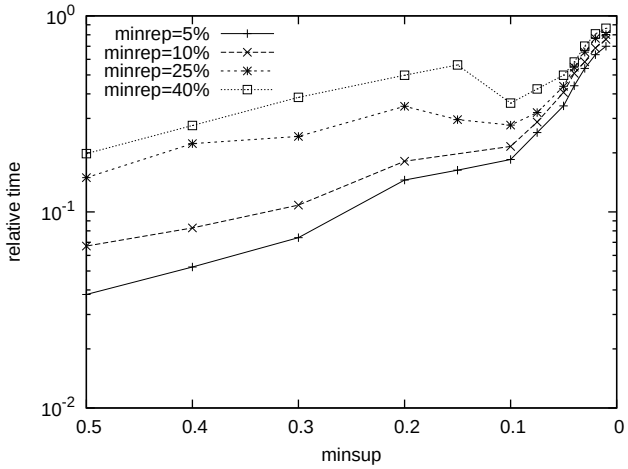
Some interesting frequent sets emerged from mining the Eurosong data. I used both this version of the set, and another one with (only) 50 tuples and 1849 (!) attributes for each country-country combination (representing points given). Among others I found that the United Kingdom is rather popular. The itemset corresponding to the UK getting points from another country participating in the same year, or any country after the rules changed in 2004 (notice the need for accounting for nulls) is 67.8%. On the other part of this spectrum we find Finland, for which the itemset corresponding to not getting any points from other participating countries is 70.7%. (Note that this dataset did not yet include the results from the 2006



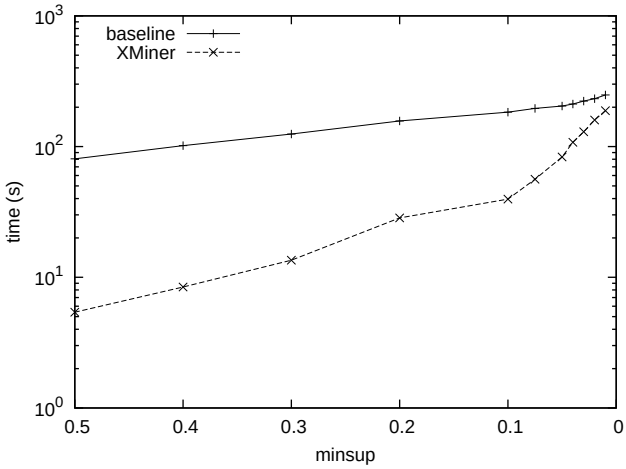
(a)



(b) minrep=10%



(c)



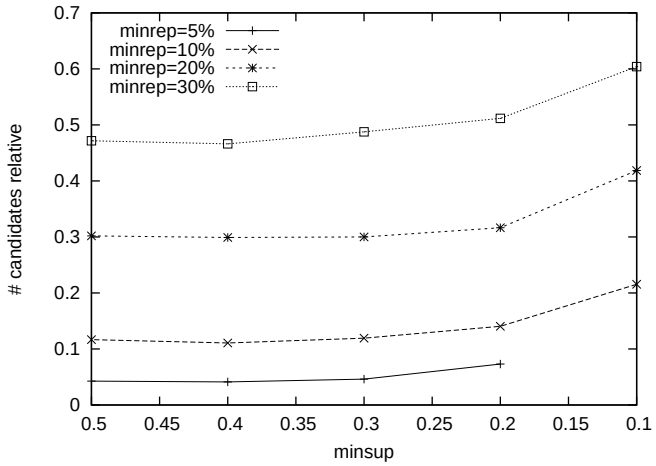
(d) minrep=10%

Figure 5.1: Sick dataset experiments

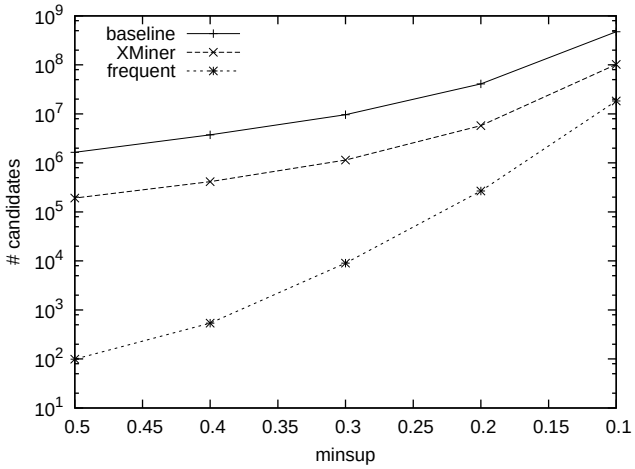
edition of the contest, when Finland actually won with a 'monster score'.) Examples from the other dataset are the following. If all three Scandinavian countries Denmark, Norway and Sweden all enter the the competition in the same year, the support of Denmark giving points to Norway, Norway giving points to Sweden, and Sweden giving points to Denmark is 30%. The support of Spain and Andorra giving each other points is even 100% (after inspection of the database, these scores turned out to be maximal), however Andorra has only started entering the competition in 2004, making this itemset not very representative.

5.3 Census Income Dataset

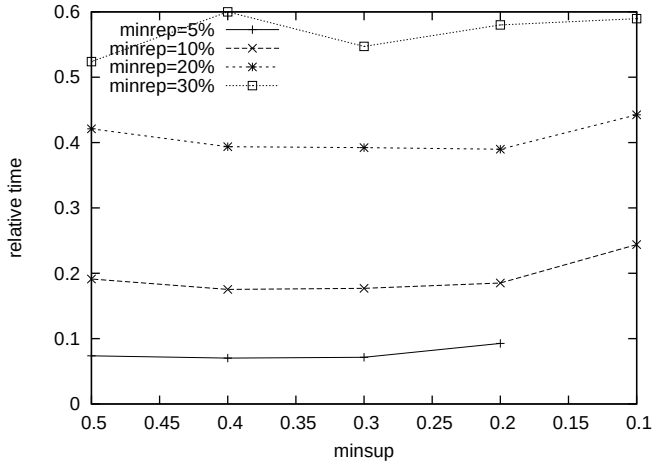
The Census database, being the largest one with nearly 200k tuples, was mined more extensively. It was mined for varying minsup, minrep, and degree of missingness in the database. First, minrep was fixed at 15% and minsup varied between %50 and 10%, and the original database was used (figures 5.3(a) and 5.3(b)). Only for a minsup of 50% and 40% did the baseline algorithm finish within 4000 seconds. Still, from this data alone we can already see the phenomenal difference with XMiner, and also the efficiency of XMiner by itself, the number of generated candidate itemsets follows the frequent ones very closely. Second, minsup was fixed at 15% and experiments were run for different minreps (figures 5.3(c) and 5.3(d)). To stay within reasonable time, Census 3 database was used (the one with the most values removed). For a decreasing minrep the number of frequent and representative itemsets does not increase dramatically. The number of generated candidate itemsets by XMiner increases with a similar slope, while the baseline algorithm seems to generate increasingly more candidates. This is visible in both execution time and number of generated itemsets. Finally the degree of missingness in the Census database was varied, by incrementally removing certain values of attributes (figures 5.3(e) and 5.3(f)). Due to the lack of data from the baseline algorithm for most of these datasets, I compared the number of candidate itemsets of XMiner versus the required number of frequent and representative itemsets in the output. All graphs are rather entangled, but the number of excessively generated sets always seems to remain in the 10% to 30% region.



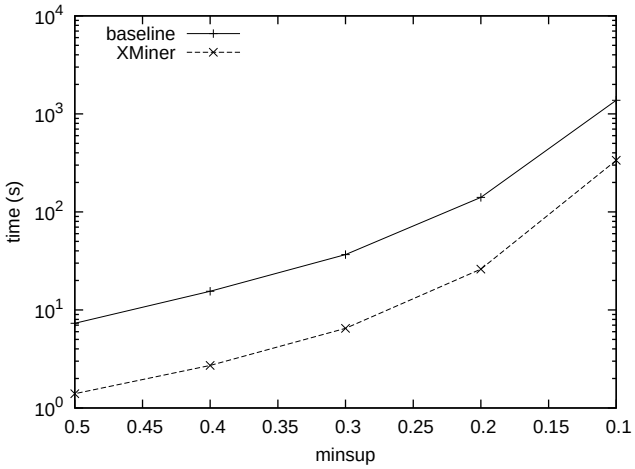
(a)



(b) minrep=10%

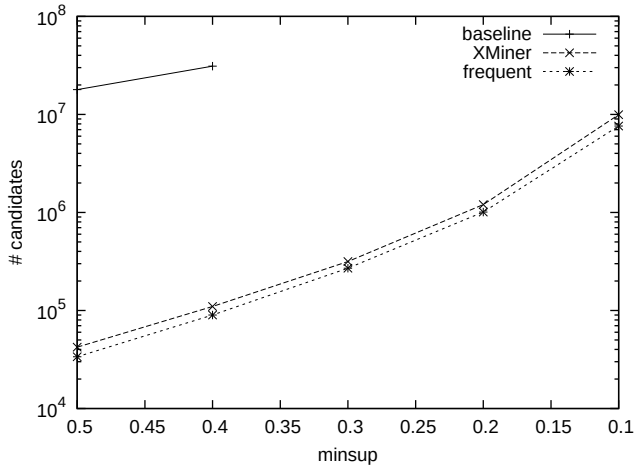


(c)

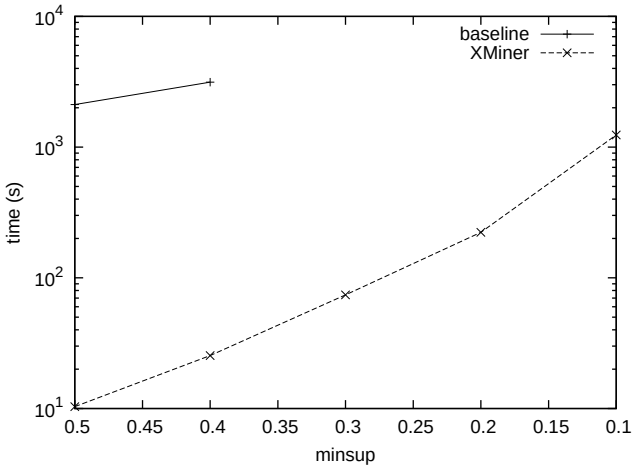


(d) minrep=10%

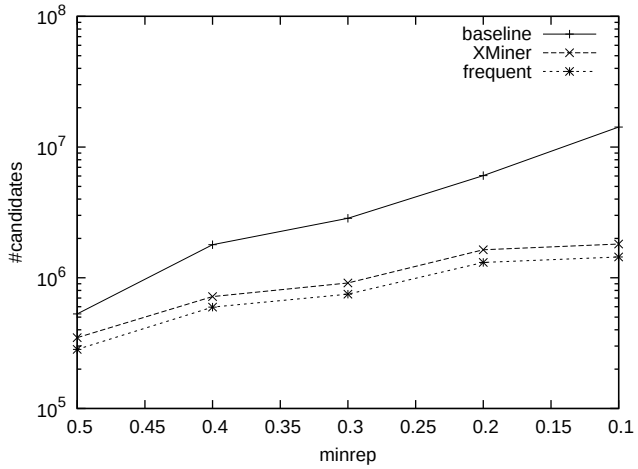
Figure 5.2: Eurosong dataset experiments



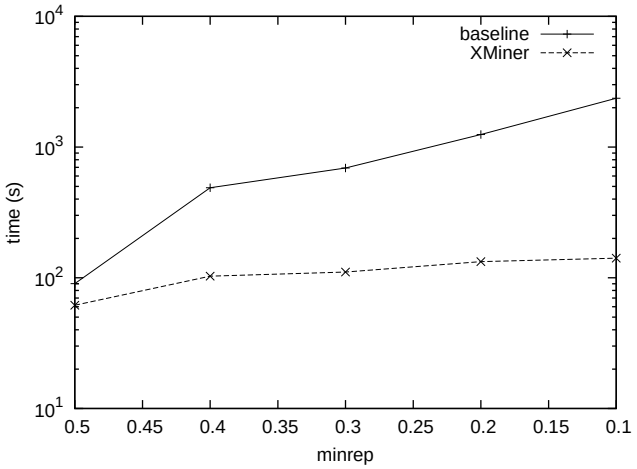
(a) minrep=15%



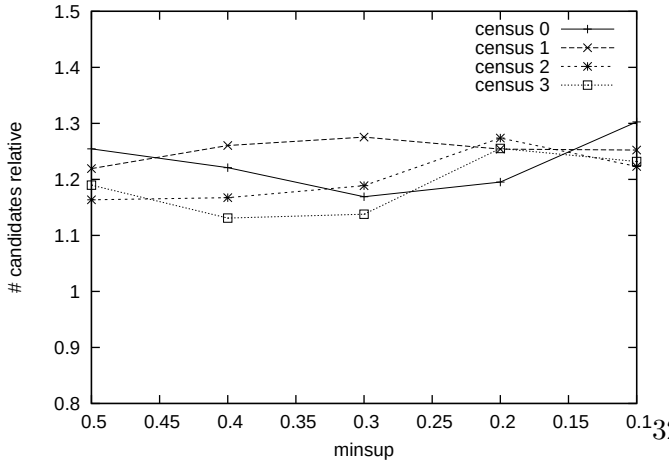
(b) minrep=15%



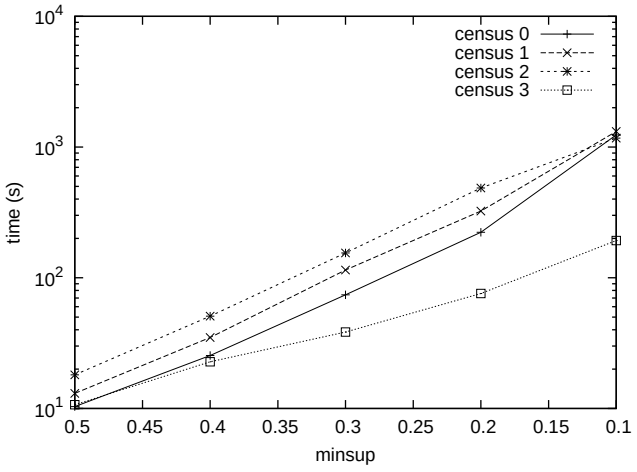
(c) minsup=15%



(d) minsup=15%



(e) minrep=15%



(f) minrep=15%

Figure 5.3: Census dataset experiments

Chapter 6

Conclusions

In this work I researched association rule mining on databases with missing values. Since the conventional measures and algorithms were defined for datasets without nulls, the usual approach is to preprocess the dataset in order to obtain a complete one. This has however a negative effect on the results.

Therefore new itemset and association rule measures were introduced [8], that are almost obvious but seem to work very well. In particular support and confidence were extended and a new measure, representativity, was introduced. Unfortunately the new support is not monotone, and confidence cannot be expressed in terms of support anymore. To overcome the first problem I introduced the extensibility property for itemsets, that can replace the role of support in an algorithm, because it is monotone and linked to support itself. I implemented this in the XMiner algorithm, which was shown to be efficient when compared to a baseline algorithm through experimentation. It is a generally applicable algorithm in that it reverts back to the Eclat algorithm if the input database contains no nulls. A basic solution for the rule generation problem was also provided. For this, generalized itemsets containing attribute items, have to be mined as well as normal itemsets, some of them might even be infrequent. By making use of the monotonicity of the absolute support count of an itemset, the number of extra itemsets that need to be considered can be reduced to the sets necessary for generating confident rules.

In my opinion this approach is to be preferred over preprocessing an incomplete database, because of the straightforwardness, not too complicated implementation, and it removes the overhead of first having to do the preprocessing itself. There is however room for improvement and optimization, especially for the rule generation algorithm. This can provide for interesting future work.

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, pages 207–216, 1993.

[2] R. Agrawal, T. Imielinski, and A.N. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 478–499, 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. ACM SIGMOD*, pages 85–93, Seattle, Washington, 1998.

[5] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.

[6] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 2003. To appear.

[7] S. Hettich and S. D. Bay. The UCI KDD Archive [http://kdd.ics.uci.edu] Irvine, CA: University of California, Dept. of inf. and comp. science, 1999.

[8] A. Ragel and B. Crémillieux. Treatment of missing values for associatio rules. In *Research and Development in Knowledge Discovery and Data Mining*, LNAI, 1998.

[9] D. Rubin. Inference and missing data. *Biometrika*, 63:581–592, 1976.

[10] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 432–444, 1995.

[11] R. Srikant and R. Agrawal. Mining generalized association rules. pages 407–419, 1995.

[12] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. 2001.

[13] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of the Third International Conference on Knowledge Discovery in Databases and Data Mining*, pages 283–286, 1997.

[14] M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May/June 2000.

[15] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2003.