Involving Aggregate Functions in Multi-relational Search

Arno J. Knobbe[1,2], Arno Siebes[2] , and Bart Marseille[1]

[1]Kiminkii,
Vondellaan 160, NL-3521 GH Utrecht, The Netherlands
a.knobbe@kiminkii.com,
b.marseille@kiminkii.com
[2]Utrecht University,
P.O. box 80 089, NL-3508 TB Utrecht, The Netherlands
siebes@cs.uu.nl

Abstract The fact that data is scattered over many tables causes many problems in the practice of data mining. To deal with this problem, one either constructs a single table by propositionalisation, or uses a Multi-Relational Data Mining algorithm. In either case, one has to deal with the non-determinacy of one-to-many relationships. In propositionalisation, aggregate functions have already proven to be powerful tools to handle this non-determinacy. In this paper we show how aggregate functions can be incorporated in the dynamic construction of patterns of Multi-Relational Data Mining.

1 Introduction

This paper presents a new paradigm for dealing with multi-relational data which involves aggregate functions. In [5, 6] the potential of these aggregate functions was demonstrated. In those papers, the aggregate functions were computed statically during a propositionalisation phase. Most MRDM algorithms compute new candidates dynamically. In our approach we combine the dynamic nature of MRDM with the power of aggregate functions by computing them on the fly.

The presented paradigm is centered around a new pattern-language that relies heavily on the use of aggregate functions. These aggregate functions are a powerful and generic way of dealing with the non-determinacy which is central to the complexity of Multi-Relational Data Mining (MRDM). Like many of the state-of-the-art MRDM approaches, we will use a top-down search that progressively 'discovers' relevant pieces of substructure by involving more tables. However, whenever a new table is involved over a one-to-many association, a range of aggregate functions (including the well-known existential expressions) can be applied to capture features of the local substructure. The proposed pattern language supports a mixture of existential expressions and aggregates.

The algorithms in [5,6] show that aggregate functions provide a unique way of characterizing groups of records. Rather than testing for the occurrence of specific

T. Elomaa et al. (Eds.): PKDD, LNAI 2431, pp. 287-298, 2002.

records in the group, which is the predominant practice in traditional algorithms, typically the group as a whole is summarized. It turns out that these algorithms work well on databases that combine a high level of non-determinacy with large numbers of (numeric) attributes.

Although producing good results, these propositionalisation approaches have some disadvantages. These are due to the fact that all multi-relational features are constructed statically during a preprocessing stage, before the actual searching is done. In contrast, most MRDM algorithms select a new set of relevant features dynamically, based on a subset of examples under investigation, say at a branch in a decision tree. Our approach overcomes the limitations of the propositionalisation approach, by introducing aggregate functions dynamically.

The work in this paper builds on a previously published MRDM framework [3]. The pattern language in this framework (Selection Graphs) will be extended to deal with our need for aggregation. Edges in the graph, which originally represented existential constraints, are annotated with aggregate functions that summarise the substructure selected by the connected sub-graph. Moreover, in [3] we also present a number of multi-relational data mining primitives in SQL. These primitives can be used to get statistics for a set of refinements using a single query to a database. Such primitives are an essential ingredient of a scalable data mining architecture. Section 5 presents a new collection of primitives to support the new constructs in our extended pattern language. Data mining primitives are especially important for the presented paradigm, as information about appropriate refinements involving aggregate functions can often only be obtained from the data, not from domain knowledge.

The structure of this paper is as follows. In Section 2 we introduce aggregate functions and give some important properties. In the next section we generalize Selection Graphs to include aggregate functions. The fourth section discusses the refinement operations on these Generalised Selection Graphs (GSG) and in Section 5 we show how these refinements can be tested using SQL. Our experimental results are given in Section 6. Our conclusions are formulated in the final section of this paper.

2 Aggregate Functions

Aggregate functions are functions on sets of records. Given a set of records and some instance of an aggregate function, we can compute a feature of the set which summarizes the set. We will be using aggregates to characterize the structural information that is stored in tables and associations between them. Consider two tables P and Q, linked by a one-to-many association A. For each record in P, there are multiple records in Q, and A thus defines a grouping over Q. An aggregate function can now be used to describe each group by a single value. Because we have a single value for every record in P, we can think of the aggregate as a virtual attribute of P. Given enough aggregate functions, we can characterize the structural information in A and Q. Note that, depending on the multiplicity of A, there may be empty groups in Q. These will be naturally treated by aggregates such as *count*. Other aggregates, such as *min*, will produce NULL values.

Aggregate functions typically (but not necessarily) involve not only the structural information in A, but also one or more attributes of Q. In theory we could even consider aggregate functions that include further tables connected to Q. For every association, we could thus define an infinite amount of aggregate functions. In order to obtain a manageable set of refinements per hypothesis, we will restrict the set of aggregate functions to the following list (all of which are part of standard SQL): *count*, *count distinct*, *min*, *max*, *sum* and *avg*. These functions are all real-valued, and work on either none or one attribute. It should be noted that we can still define large numbers of aggregate functions with this restricted list, by putting conditions on records in Q. However, these complex aggregate functions are discovered by progressive refinements.

Because we will consider varying sets of records during our search by adding a range of conditions, we will need to examine how the value of aggregate functions depends on different sets of records. The following observations will be relevant for selecting proper refinements (see section 4).

Definition 1.

- An aggregate function f is *ascending* if, given sets of records S and S', $S' \subseteq S \Rightarrow f(S') \leq f(S)$.
- An aggregate function f is *descending* if, given sets of records S and S', $S' \subseteq S \Rightarrow f(S') \geq f(S)$.

Lemma 1.

- Aggregate functions *count*, *count distinct* and *max* are ascending.
- Aggregate function *min* is descending.
- Aggregate functions *sum* and *avg* are neither ascending nor descending.

Example 1. In the mutagenesis problem database [8], there are tables for the concepts *molecule*, *atom* and *bond*. Promising aggregate functions to describe the relation between *molecule* and *atom* are: `count(atom)`, `min(atom.charge)`, `avg(atom.charge)` etc. We might also have similar functions for C-atoms in a molecule, or for C-atoms involved in a double bond. Clearly the count decreases if we only consider a subset of atoms, such as the C-atoms. In contrast, the minimum charge will increase.

3 Generalized Selection Graphs

We introduced Selection Graphs (SG) in [3, 4] as a graphical description of sets of objects in a multi-relational database. Objects are covered by a Selection Graph if they adhere to existential conditions, as well as attribute conditions. Every node in the graph corresponds to a table in the data model, and every edge corresponds to an association. If an edge connects two nodes, the corresponding association in the data model will connect two tables that correspond to the two nodes. The graphical structure of the Selection Graph thus reflects that of the underlying data model, and consequently the set of possible Selection Graphs is determined by the data model. For a structured example to be covered by a given SG, it will have to exhibit the same graphical structure as the SG. Next to the graphical conditions, an SG also holds

attribute conditions. Every node contains a, possible empty, set of conditions on the attributes of the associated table. Finally, the target table in the data model, which identifies the target concept, appears at least once in any SG.

As [3] demonstrates, there is a simple mapping from Selection Graphs to SQL. A similar mapping to first-order logic exists. However, with these mappings we loose the intuitive graphical nature of the link between pattern language and declarative bias language. Selection Graphs are simply an intuitive means of presentation because of their close link with the underlying data model. Secondly, the refinement steps in the mining process are simple additions of the SGs. We will show how aggregate functions are a natural generalization of the existential conditions represented by edges in an SG, which makes Selection Graphs an ideal starting point for our approach.

To support aggregate functions with SGs we have to extend the language with a selection mechanism based on local structure. In particular, we add the possibility of *aggregate conditions,* resulting in Generalized Selection Graphs (GSG).

Definition 2. An *aggregate condition* is a triple (f, o, v) where f is an aggregate function, o a comparison operator, and v a value of the domain of f.

Definition 3. A *generalized selection graph* is a directed graph (N, E), where N is a set of triples (t, C, s), t is a table in the data model and C is a, possibly empty, set of conditions on attributes in t of type *t.a operator c*; the *operator* is one of the usual comparison operators such as = and >. The flag s has the possible values *open* and *closed*. E is a set of tuples (p, q, a, F) where $p, q \in N$ and a is an association between $p.t$ and $q.t$ in the data model. F is a non-empty set of aggregate conditions with the same aggregate function. The generalized selection graph contains at least one node n_0 (the *root node*) that corresponds to the target table t_0.

Generalized Selections Graphs are simply SGs with two important extensions. First, the use of edges to express existential constraints is generalized by adding an aggregate condition. Secondly, nodes may be either *open* or *closed*, which determines the possible refinements (see next section). A given Generalized Selection Graph (GSG) can be interpreted as follows. Every node n represents a set of records in a single table in the database. This set is governed by the combined restriction of the set of attribute conditions C and the aggregate conditions produced by each of the subgraphs connected to n. Each subgraph represents a similar set of records, which can be turned into a virtual attribute of n by the aggregate function. We thus have a recursive definition of the set of records represented by the root node. Each record corresponds to one example that is covered by the GSG. We will use the same recursive construct in a translation procedure for SQL, which is addressed in section 5.

Note that the simple language of Selection Graphs is contained in GSG. The concept of a selection edge [3] is replaced by the equivalent aggregate condition (*count*, >, 0). In fact, we will use the directed edge without any aggregate condition as a purely syntactic shorthand for this aggregate condition.

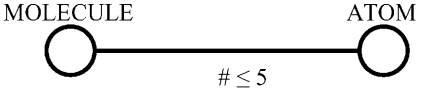
4 Refinements

The majority of ILP and MRDM algorithms traverse a search space of hypotheses which is ordered by θ-subsumption [1,9]. This ordering provides a syntactic notion of generality: by simple syntactic operations to hypotheses we can derive clauses that are more specific. The two basic forms of operation are:

- add a condition to an existing node in the Selection Graph (roughly corresponds to a substitution in FOL).
- add a new node to the selection graph (add a literal to the clause).

These so-called refinements are a prerequisite to the top-down search which is central to so many Data Mining algorithms. This approach works smoothly under the following (often implicit) assumption: clause c is at least as general as clause c' if c θ-subsumes c'. That is , θ-subsumption is a good framework for generality. This assumption holds for the Selection Graphs, introduced in [3]. However, this is not the case for the Generalized Selection Graphs that we are considering in this paper. Simple substitutions of the above mentioned style do not necessarily produce more specific patterns.

Example 2. Consider the molecular database from example 1. Assume we are refining a promising hypothesis: the set of molecules with up to 5 atoms.



A simple substitution would produce the set of molecules with up to 5 C-atoms. Unfortunately, this is a superset of the original set. The substitution has not produced a specialization.

The problem illustrated by example 2 lies with the response of the aggregate condition to changes of the aggregated set resulting from added attribute conditions. Refinements to a subgraph of the GSG only work well if the related aggregate condition is *monotone*.

Definition 4. An aggregate condition (f, o, v) is *monotone* if, given sets of records S and S', $S' \subseteq S$, $f(S')\ o\ v \Rightarrow f(S)\ o\ v$.

Lemma 2.

- Let A = (f, ≥, v) be an aggregate condition. If f is ascending, then A is monotone.
- Let A = (f, ≤, v) be an aggregate condition. If f is descending, then A is monotone.
- The following aggregate conditions are monotone: (*count*, ≥, v), (*count distinct*, ≥, v), (*max*, ≥, v), (*min*, ≤, v).

Lemma 2 shows that only a few aggregate conditions are safe for refinement. A subgraph of the GSG that is joined by a non-monotone aggregate condition should be left untouched in order to have only refinements that reduce the coverage. The flag s at each node stores whether nodes are safe starting points for refinements. *Closed* nodes will be left untouched. This leads us to the following collection of refinements:

- **add attribute condition.** An attribute condition is added to *C* of an *open* node.
- **add edge with aggregate condition.** This refinement adds a new edge and node to an *open* node according to an association and related table in the data model. The set *F* of the new edge contains a single aggregate condition. If the aggregate condition is monotone, the new node is *open*, and *closed* otherwise.
- **add aggregate conditions.** This refinement adds an aggregate condition to an existing edge. If any of the aggregate conditions is non-monotone, the related node will be *closed*, otherwise *open*.

Theorem 1. The operators **add attribute condition**, **add edge with aggregate condition**, and **add aggregate conditions** are proper refinements. That is, they produce less general expressions.

The above-mentioned classes of refinements imply a manageable set of actual refinements. Each *open* node offers opportunity for refinements by each class. Each attribute in the associated table gives rise to a set of **add attribute conditions**. Note that we will be using primitives (section 5) to scan the domain of the attribute, and thus naturally treat both nominal and numeric values. Each association in the data model connected to the current table gives rise to **add edge with aggregate conditions**. Our small list of SQL aggregate functions form the basis for these aggregate conditions. Finally, existing edges in the GSG can be refined by applying **add aggregate condition**.

5 Database Primitives

The use of database primitives for acquiring sufficient statistics about the patterns under consideration is quite widespread in KDD. A set of multi-relational primitives related to the limited language of Selection Graphs was given in [3]. The purpose of these primitives was to compute the support for a set of hypotheses by a single query to the database. These counts form the basis for the interestingness measures of choice, related to a set of refinements. A single query can for example assess the quality of an attribute condition refinement of a given Selection Graph, for all possible values of the attribute involved. The counts involved are always in terms of numbers of examples, i.e. records in the target table.

An elegant feature of the primitives related to Selection Graph is that they can be expressed in SQL using a single `SELECT` statement. In relational algebra this means that every primitive is counting over selections on the Cartesian product of the tables involved. Both the joins represented by selection edges, as well as the attribute conditions appear as simple selections. This works well for Selection Graphs because they describe combinations of particular records, each of which appears as a tuple in the Cartesian product. The Generalized Selection Graphs introduced in this paper are about properties of *groups* of records rather than the occurrence of individual combinations of records. Therefore the primitives for GSGs are computed using nested selections.

We start by introducing a basic construct which we will use to define our range of primitives. Every node in a GSG represents a selection of records in the associated

table. If we start at the leafs of the GSG and work back to the root, respecting all the selection conditions, we can compute the selection of records in the target table. This is achieved as follows. First we produce a list of groups of records in a table *Q* at a leaf node by testing on the aggregate condition. Each group is identified by the value of the foreign key:

```
SELECT foreign-key
FROM Q
WHERE  attribute-conditions
GROUP  BY foreign-key
HAVING aggregate-conditions
```

We then join the result *Q'* with the parent table *P* to obtain a list of records in *P* that adheres to the combined conditions in the edge and leaf node :

```
SELECT P. primary-key
FROM P, Q'
WHERE P. primary-key = Q'. foreign-key
```

This process continues recursively up to the root-node, resulting in a large query of nested `SELECT` statements. The second query can be extended with the grouping construct of the first query for the next edge in the sequence. This results in exactly one `SELECT` statement per node in the GSG. This process is formalised in figure 1. The different primitives are all variations on this basic construct *C*.

5.1 CountSelection

The CountSelection primitive simply counts the number of examples covered by the GSG:

```
SELECT  count(*)
FROM C
```

5.2 Histogram

The Histogram primitive computes the distribution of values of the class attribute within the set of examples covered by the GSG.

```
SELECT  class, count(*)
FROM C
GROUP  BY class
```

5.3 NominalCrossTable

The NominalCrossTable primitive can be used to determine the effect of an attribute refinement involving a nominal attribute on the distribution of class values. All pairs of nominal values and class values are listed with the associated count. As the nominal attribute typically does not appear in the target table, we cannot rely on the basic construct to produce the NominalCrossTable. Rather, we will have to augment the basic construct with bookkeeping of the possible nominal values.

```
SelectSubGraph (node n)
S = 'SELECT ' + n.Name( ) + '. '
if (n.IsRootNode( ))
        S.add (n.PrimaryKey( ))
else
        S.add (n.ForeignKey( ))
S.add ('FROM ' + n.Name( ))
for each child i of n do
        S_i = SelectSubGraph(i)
        S.add (', ' + S_i + 'S' + i )
S.add ('WHERE ' + n.AttributeConditions( ))
if (! n.IsLeaf( ))
        for each child i of n do
                S.add ('AND ' + n.Name( ) + '. ' + n.PrimaryKey( ) +
                '= S' + i + '. ' + i.ForeignKey( ))
if (! n.IsRootNode( ))
        S.add ('GROUP BY ' + n.Name( ) + '. ' + n.ForeignKey( ))
        S.add ('HAVING ' + n.ParentEdge( ). AggregateCondition( ))
Return S
```

Fig. 1. The select-subgraph algorithm

The idea is to keep a set of possible selections, one for each nominal value, and propagate these selections from the table that holds the nominal attribute, up to the target table. These selections all appear in one query, simply by grouping over the nominal attribute in the query which aggregates the related table. The basic construct is extended, such that an extra grouping attribute *X* is added to all the nested queries along the path from the rood-node to the node which is being refined:

```
SELECT X, foreign-key
FROM Q
GROUP BY X, foreign-key
HAVING aggregate-conditions
```

The extra attribute in this query will reappear in all (nested) queries along the path to the rood-node. The actual counting is done in a similar way to the Histogram:

```
SELECT X, class, count(*)
FROM C
GROUP BY X, class
```

5.4 NumericCrossTable

The NumericCrossTable primitive can be used to obtain a list of suitable numeric refinements together with the effect this has on the class-distribution. It is very similar to the NominalCrossTable, but requires yet a little more bookkeeping. This is because a given record may satisfy a number of numeric conditions, whereas it will only satisfy a single nominal condition. The process starts by producing a list of combinations of records and thresholds, and then proceeds as before. Note that there is a version of the NumericCrossTable for each numeric comparison operator.

```
SELECT b.X, a.foreign-key
FROM Q a, (SELECT DISTINCT X FROM Q) b
WHERE a.X ≤ b.X
GROUP BY b.X, a.foreign-key
HAVING aggregate-conditions
```

5.5 AggregateCrossTable

The AggregateCrossTable is our most elaborate primitive. It can be used to obtain a list of suitable aggregate conditions for a given aggregate function. The AggregateCrossTable primitive demonstrates the power of data mining primitives because the list of suitable conditions can only be obtained through extensive analysis of the database. It cannot be obtained from the domain knowledge or a superficial scan when starting the mining process.

We will treat candidate aggregate conditions as virtual attributes of the parent table. A first query is applied to produce this virtual attribute. Then the NumericCrossTable primitive is applied with the virtual attribute as the candidate. The first step looks as follows:

```
SELECT foreign-key, aggregate-function
FROM Q
GROUP BY foreign-key
```

6 Experiments

The aim of our experiments is to show that we can benefit from using a MRDM framework based on GSG rather than SG. If we can prove empirically that substantial results can be obtained by using a pattern-language with aggregate functions, then the extra computational cost of GSG is justified.

The previous sections propose a generic MRDM framework that is independent of specific algorithms. In order to perform our experiments however, we have implemented a rule-discovery algorithm that produces collections of rules of which the body consists of GSG and the head is a simple condition on the target attribute. The discovery algorithm offers a choice of rule evaluation measures as proposed in [7] of which two (Novelty and Accuracy) have been selected for our experiments. Rules are ranked according to the measure of choice.

6.1 Mutagenesis

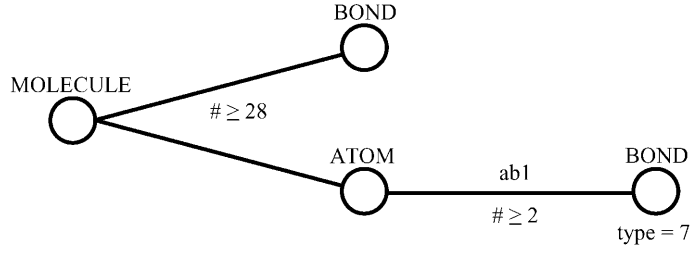
For our first experiment, we have used the Mutagenesis database with background 'B3' [8]. This database describes molecules falling in two classes, *mutagenic,* and *non-mutagenic.* The description consists of the atoms and the bonds that make up the compound. In particular, the database consists of 3 tables that describe directly the graphical structure of the molecule (**molecule**, **atom**, and **bond**). We have used the so-called 'regression-friendly' dataset for B3 that consists of 188 molecules, of which 125 (66.5%) are *mutagenic*.

Table 1. Summary of results for Mutagenesis

		Novelty		Accuracy	
		GSG	SG	GSG	SG
Best	Nov.	0.173	0.146	0.121	0.094
	Acc.	0.948	0.948	1.0	1.0
	Cov.	115	97	68	53
Top 10	Nov.	0.170	0.137	0.115	0.088
	Acc.	0.947	0.914	1.0	1.0
	Cov.	113.2	103.4	64.6	49.6

Table 1 presents a summary of our results. The left half gives the results when optimizing for Novelty. The right half does this for Accuracy. The top half gives the measures for the best rule, whereas the bottom half gives the averages for the best 10 rules. We see that all results for GSG are better than those of SG. In those cases were both SG and GSG have accuracy 1.0, which cannot be bettered, GSG has a far larger coverage.

A typical example of a result is given below. This GSG describes the set of all molecules that have at least 28 bonds as well as an atom that has at least two bonds of type 7. This result cannot be found with standard MRDM algorithms because of the condition of at least 28 bonds. It would also be difficult for the propositionalisation approaches because there first is a condition on the existence of an atom and then an aggregate condition on bonds of type 7 of that atom.



6.2 Financial

Our second experiment concerns the Financial database, taken from the Discovery Challenge organised at PKDD '99 and PKDD 2000 [10]. The database consists of 8 tables describing the operations of customers of a Czech bank. Among the 682 customers with a loan, we aim to identify subgroups with a high occurrence of bad

loans. Overall, 76 loans (11.1%) are bad. For all rules a minimum of 170 examples (25%) was required.

Table 2 summarizes the results obtained for Financial. Clearly, all results for GSG are significantly better than those for SG, in terms of both Novelty and Accuracy. Admittedly, the results were obtained with rules of lower Coverage, but this was not an optimisation criterion.

Table 2. Summary of results for Financial

		Novelty		Accuracy	
		GSG	SG	GSG	SG
Best	Nov.	0.052	0.031	0.051	0.025
	Acc.	0.309	0.190	0.314	0.196
	Cov.	178	268	172	199
Top 10	Nov.	0.050	0.027	0.047	0.023
	Acc.	0.287	0.178	0.297	0.185
	Cov.	194.5	281.8	171.4	208.3

7 Conclusion

In this paper we have presented a new paradigm for Multi-Relational Data Mining. The paradigm depends on a new way of capturing relevant features of local substructure in structured examples. This is achieved by extending an existing pattern language with expressions involving so-called aggregate functions. Although such functions are very common in relational database technology and are provided by all RDBMSes, they have only sporadically been used in a MRDM context. This paper shows that this is unfortunate and that aggregates are a powerful as well as natural way of dealing with the complex relationships between components of the structured examples under investigation. As our treatment concerns a generic framework based on an extended pattern language, it can be used to derive a range of top-down mining algorithms. We have performed experiments with a single rule discovery algorithm, but other approaches are clearly supported.

We have illustrated the usefulness of our paradigm on two well-known database, Mutagenesis [8] and Financial [10]. The experimental results show a significant improvement over traditional MRDM approaches. Moreover, the resulting patterns are intuitive and easily understandable. The positive results suggest future research into the benefits of non-top-down search strategies to overcome the current exclusion of non-monotone refinements.

References

1. Džeroski, S., Lavrač, N., An Introduction to Inductive Logic Programming, In [2]
2. Džeroski, S., Lavrač, N., *Relational Data Mining*, Springer-Verlag, 2001
3. Knobbe, A. J., Blockeel, H., Siebes, A., Van der Wallen, D.M.G. *Multi-Relational Data Mining,* In Proceedings of Benelearn '99, 1999

4. Knobbe, A. J. Siebes A, Van der Wallen D.M.G., *Multi-Relational Decision Tree Induction*. In proceedings of PKDD'99, LNAI 1704, pp. 378-383, 1999
5. Knobbe A. J., De Haas, M., Siebes, A., *Propositionalisation and Aggregates,* In Proceedings of PKDD 2001, LNAI 2168, pp. 277-288, 2001
6. Krogel, M. A., Wrobel, S., *Transformation-Based Learning Using Multi-relational Aggregation*, In Proceedings of ILP 2001, LNAI 2157, pp. 142-155. 2001
1. Lavrač, N., Flach, P., Zupan, B., *Rule Evaluation Measures: A Unifying View*, In Proceedings of ILP '99, 1999
8. Srinivasan, A., King, R.D., Bristol, D.W., *An Assessment of ILP-Assisted Models for Toxicology and the PTE-3 Experiment*, In Proceedings of ILP '99, 1999
9. Van Laer, W., De Raedt, L., How to Upgrade Propositional Learners to First Order Logic: A Case Study, In [2]
10. Workshop notes on Discovery Challenge PKDD '99, 1999