Refining Aggregate Conditions in Relational Learning

Celine Vens, Jan Ramon, and Hendrik Blockeel

Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, 3001 Leuven, Belgium
{celine.vens, jan.ramon, hendrik.blockeel}@cs.kuleuven.be

Abstract. In relational learning, predictions for an individual are based not only on its own properties but also on the properties of a set of related individuals. Many systems use aggregates to summarize this set. Features thus introduced compare the result of an aggregate function to a threshold. We consider the case where the set to be aggregated is generated by a complex query and present a framework for refining such complex aggregate conditions along three dimensions: the aggregate function, the query used to generate the set, and the threshold value. The proposed aggregate refinement operator allows a more efficient search through the hypothesis space and thus can be beneficial for many relational learners that use aggregates. As an example application, we have implemented the refinement operator in a relational decision tree induction system. Experimental results show a significant efficiency gain in comparison with the use of a less advanced refinement operator.

1 Introduction

In relational learning, predictions for an individual are based not only on its own properties but also on the properties of a set of related individuals. Many systems use aggregates to summarize this set. Features are then constructed by comparing the result of the aggregate function to a threshold, we call such a feature an aggregate condition. For example, in the context of a data set on parents and children, a possible feature could be *"the maximum age of the person's children is larger than 10"*.

Many learning systems rely on a general-to-specific ordering of the hypotheses to traverse the hypothesis space in an efficient way. Only few of the existing systems that learn hypotheses with aggregates extrapolate this generality ordering to the aggregate conditions, and when they do, they do it in a restricted way. For instance, the feature just mentioned could be refined in three ways: by changing the aggregate function (e.g., change *maximum* into *average*), the subset aggregated over (e.g., specialize *children* into *daughters*) or the threshold compared with (e.g., increase *10* to *15*). No current relational learners consider all three kinds of refinements, and indeed the effect of such refinements on the generality of a rule, and the interaction between these effects, are non-trivial and have not been studied up till now.

J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.): PKDD 2006, LNAI 4213, pp. 383–394, 2006.

This paper presents the first comprehensive study of these effects and interactions. This study leads to the description of a refinement operator that enables relational learners to traverse the hypothesis space more efficiently.

The paper is organized as follows. We start by describing several approaches to learning that use aggregates in Sect. 2. Then we elaborate on the monotonicity properties of aggregate conditions and present a general framework for specializing them (Sect. 3). In Sect. 4, we apply the framework to a relational decision tree learner and assess the efficiency gain that this results in. Finally, conclusions and ideas for further research are presented in Sect. 5.

2 Related Work

Perhaps the most closely related work, at first sight, is the research started by Ng et al. [1] on finding frequent itemsets that fulfill constraints with aggregations (for instance, in addition to minimal support, one can impose that the average price of the items in an itemset must be above some threshold). That research also involves studies of the monotonicity properties of aggregates, but the goals are different from ours: the aggregates occur in the constraints defining the hypothesis space rather than in the learned hypotheses; they summarize information over attributes of one entity rather than over different entities; etc. As a result also the methods developed are very different.

On the other hand, there is work that shares our goal of learning hypotheses with aggregates. Here we can distinguish methods that use a fixed set of aggregates defined in advance [2,3,4,5], and methods that construct the aggregates as part of the learning process. The latter, on which we will focus, are especially useful when one wants to consider complex aggregates, where the set to be aggregated over is generated by a complex query: there may be too many such aggregates to compute and store them all during preprocessing. Perlich and Provost [6] provide a detailed examination of aggregation in relational learning and demonstrate that such complex aggregate conditions can significantly improve generalization performance.

Unfortunately, finding good hypotheses with complex aggregate conditions is difficult. Besides the fact that the hypothesis space is significantly expanded by allowing complex aggregate conditions, it also becomes more difficult to search this space in a structured way, because the effect of refinements of the aggregate condition on the generality of the hypothesis is not well-understood in general. As a result, all current relational learners are somehow limited with respect to the aggregates they can learn. Krogel and Wrobel [7] introduce in their propositionalized table aggregate functions that apply not only to single attributes, but also to pairs of attributes, one of which has to be nominal and serves as a *group by* condition. The resulting aggregate conditions are still of limited complexity and are not refined further during the search. Knobbe et al. [8] propose a method for subsequently specializing the set to be aggregated. By restricting the application of this specialization operator to aggregate functions where its effect is well-understood, they can search the hypothesis space in a general-to-specific way, but this obvi-

ously limits the kind of complex conditions that can be found. Van Assche et al. [9] discuss refinement of aggregate conditions in the context of relational decision tree induction. They handle the expansion of the feature space by upgrading the tree learner into a random forest induction system, where random feature subsampling reduces the number of features tested at each node in a tree. Uwents and Blockeel [10] describe relational neural networks as a subsymbolic approach towards learning complex aggregates. Their approach is not constrained to using predefined aggregate functions and does not make a distinction between searching for aggregate functions and searching for complex conditions, but the resulting theories are also not interpretable in terms of well-understood aggregates and conditions.

Clearly none of the existing approaches fully solve the problem of searching general-to-specific in a hypothesis space that may include aggregates of arbitrary complexity. This paper presents the first solution to it.

3 Specializing Aggregate Conditions

ILP systems (inductive logic programming [11]) learn sets of logic clauses, typically by learning one clause at a time. A single clause has the form $h \leftarrow b_1, b_2, \ldots, b_n$, where h and b_i are literals. The ILP system usually finds a clause by starting with the empty clause $h \leftarrow$ and gradually refining it, adding literals to the body of the clause, until it is consistent with the data. It is known that adding literals to the body cannot increase the coverage of the clause, and this can be used to prune the search space. We call refinements that never increase the coverage of a clause, valid refinements.

Now assume that one of the b_i is an aggregate condition of the form $F(\{V|Q\})$ θ R, where F is an aggregate function, Q a conjunctive query, V a variable occurring in Q, θ one of $\{\leq, \geq\}$ and R a numeric value. For instance, the clause

$$person(P, pos) \leftarrow max(\{A|child(P, C), age(C, A)\}) \geq 10$$

classifies a person as positive if the maximum age of his children is higher than 10. Such a clause could now be refined not only by extending the clause itself with a literal b_{n+1}, but also by extending the query Q, or by changing F or R. The question is then under what conditions such refinements are valid. For instance, changing the preceding aggregate condition into

$$max(\{A|child(P, C), age(C, A), male(C)\}) \geq 10$$

is a valid refinement, but with *min* instead of *max* this would not be the case. Also with $\leq$ instead of $\geq$ the above refinement would not be valid.

In this section, we will provide an answer to which refinements of aggregate conditions are valid. We start by discussing aggregate conditions in more detail (Sect. 3.1). Then we present several classes of aggregate functions with an order relation (Sect. 3.2). Afterwards, we elaborate on the refinement of aggregate conditions (Sect. 3.3). Finally, we discuss so-called refinement cubes that visualize possible refinements (Sect. 3.4). Although in this section we have chosen the ILP formalism for expressing relational concepts, the theory behind it can be applied to other relational representations as well.

3.1 Aggregate Conditions

The aggregate conditions that we consider have the following general form: $F(S) \in I$, with F an aggregate function, I a numerical interval, and S a set to be aggregated. For now we will make abstraction of the fact that S is generated by some query Q. More formally, an aggregate condition c takes the following signature: $c : \mathbb{F} \times \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{B}$, with $\mathbb{F}$ a set of aggregate functions (e.g., $\mathbb{F} = \{$count,max,min,sum,avg$\}$), $\mathbb{S}$ a set of sets, $\mathbb{I}$ a set of intervals, and $\mathbb{B}$ the set of boolean values, that indicate whether the condition $F(S) \in I$ holds.

In order to define valid refinements of aggregate conditions we need the concept of *monotonicity*.

Definition 1. *A function* $f(x_1, ..., x_n)$ *is*

- monotone *in* x_i *iff* $x_i < x_{i'} \Rightarrow f(x_1, ..., x_i, ..., x_n) < f(x_1, ..., x_{i'}, ..., x_n)$,
- anti-monotone *in* x_i *iff* $x_i < x_{i'} \Rightarrow f(x_1, ..., x_i, ..., x_n) > f(x_1, ..., x_{i'}, ..., x_n)$,
- *and* non-monotone *in* x_i *otherwise.*

To investigate the monotonicity properties of an aggregate condition $F(S) \in I$, an order relation is needed on the domains $\mathbb{F}$, $\mathbb{S}$, $\mathbb{I}$, and $\mathbb{B}$. We define these relations as follows:

- $\mathbb{F}$: $F_1 \preceq_{\mathbb{F}} F_2 \Leftrightarrow \forall S \in \mathbb{S} : F_1(S) \leq F_2(S)$, this is discussed in the next section,
- $\mathbb{S}$: $S_1 \preceq_{\mathbb{S}} S_2 \Leftrightarrow S_1 \subseteq S_2$,
- $\mathbb{I}$: $I_1 \preceq_{\mathbb{I}} I_2 \Leftrightarrow I_1 \subseteq I_2$,
- $\mathbb{B}$: *false* $\preceq_{\mathbb{B}}$ *true.*

3.2 Ordering the Aggregate Functions

In this section we discuss several parameterized classes of aggregate functions that are ordered and together cover all aggregates of interest. For this paper, we set the aggregate functions of interest to be those defined in standard SQL, i.e., *max*, *min*, *avg*, *sum*, and *count*.

Generalized Averages. We define a class of generalized averages as follows:

Definition 2

$$avg_k(S) = \left(\frac{\sum_i (x_i^k)}{n}\right)^{1/k} \text{ with } S = \{x_1, ..., x_n\}$$

The function $avg_k(S)$ is defined for $-\infty \leq k \leq \infty$ $(k \neq 0)$ if $S \subseteq \mathbb{R}^+$ and for $k = \{1, -1, \infty, -\infty, 2 * z\}$ with $z \in \mathbb{Z}$ if $S \subseteq \mathbb{R}$. If $S \subseteq \mathbb{R}^+$, then the following order relation holds: $i \leq j \Rightarrow avg_i \preceq_{\mathbb{F}} avg_j$. While this relation holds for all k, in practice only some of these k-values are commonly used: $avg_1(S) = avg(S)$, $\lim_{k \to \infty} avg_k(S) = max(S)$, and $\lim_{k \to -\infty} avg_k(S) = min(S)$. Moreover, the order relation $min \preceq_{\mathbb{F}} avg \preceq_{\mathbb{F}} max$ also holds for sets S that contain negative numbers.

Generalized Sums. For *sum* we can define an aggregate function class very similar to the generalized averages:

Definition 3

$$sum_k(S) = (\sum_i (x_i^k))^{1/k} \text{ with } 1 \leq k \leq \infty \text{ and } S = \{x_1, ..., x_n\}$$

We have that $sum_1(S) = sum(S)$ and $\lim_{k\to\infty} sum_k(S) = max(S)$. In other words, these generalized sums range from *max* to *sum*. If S contains only positive numbers, we obtain the following order relation: $i \geq j \Rightarrow sum_i \preceq_{\mathbb{F}} sum_j$.

Generalized Counts. Our last aggregate function of interest is *count*. An aggregate function that can form an aggregate class with *count* is *count distinct*. This function counts the number of *different* values in the set. We have the following order relation: $count\_distinct \preceq_{\mathbb{F}} count$.

Summary. While the proposed aggregate classes contain an infinite amount of aggregate functions, for the most important ones, we obtain the following order: $min \preceq_{\mathbb{F}} avg \preceq_{\mathbb{F}} max$, $max \preceq_{\mathbb{F}} sum$ if $S \subseteq \mathbb{R}^+$, and $count\_distinct \preceq_{\mathbb{F}} count$.

Remark that other parameterized classes for these aggregate functions exist.

3.3 Refinement of Aggregate Conditions

Having defined an ordering relation on the domains $\mathbb{F}$, $\mathbb{S}$, $\mathbb{I}$, and $\mathbb{B}$ of an aggregate condition, we can discuss the monotonicity properties of each domain and define valid refinements.

Monotonicity Properties. For the ease of explanation, we look at an aggregate condition $F(S) \in I$ as the composition of two functions:

- an aggregate function $a : \mathbb{F} \times \mathbb{S} \to \mathbb{R}$,
- a member function $m : \mathbb{R} \times \mathbb{I} \to \mathbb{B}$.

For each of the functions, we can now describe the monotonicity properties. Afterwards, we describe some issues that come along when composing them.

Monotonicity of the aggregate function. The function $a(F, S)$ is monotone in F, because the order on $\mathbb{F}$ is defined as such.

The monotonicity of $a(F, S)$ w.r.t. S depends on F. The function $a(F, S)$ is monotone in S if $F \in \{count, count\_distinct, max\}$ (if any of these functions is applied to a subset $S' \subseteq S$, its resulting value will decrease). Similarly, $a(min, S)$ is anti-monotone, and $a(sum, S)$ and $a(avg, S)$ are non-monotone in S[1].

Monotonicity of the member function. The member function $m(R, I)$ is monotone in I: decreasing the interval can cease the membership of R.

The monotonicity in R depends on I: $m(R, [v, \infty[)$ (with $v \in \mathbb{R}$) is monotone in R, $m(R,]\text{-}\infty, v])$ anti-monotone, and $m(R, [v, w])$ $(v, w \in \mathbb{R})$ non-monotone.

[1] For *sum* the monotonicity depends on the set S, e.g., if this set contains only positive numbers then $a(sum, S)$ behaves monotone.

Monotonicity of the composite function. Care is needed when composing the aggregate function and the member function. The monotonicity properties in F and S are inherited by $m(a(F,S),I)$ if this function is monotone in $a(F,S)$. However, the monotonicity in F and S is reversed if the composed function is anti-monotone in $a(F,S)$. For example, a condition $max(S) \in]\text{-}\infty, v]$ is anti-monotone in max and S.

Similarly, when the composed function is non-monotone in $a(F,S)$, the monotonicity properties in F and S are broken. For instance, for a condition $max(S) \in [v,w]$, the monotonicity properties in max and S are lost. Therefore, in the following we do not consider intervals of the form $[v,w]$, with $v,w \in \mathbb{R}$. We only consider the aggregate conditions $F(S) \leq v$ and $F(S) \geq v$.

Refinement. In order to define valid refinements for an aggregate condition, we can use the monotonicity properties described above. Keeping in mind that the goal is to obtain more specific conditions (i.e. refinements that make the condition false for some of the examples for which it was true), Definition 1 learns that a function that is monotone (anti-monotone) in one of its inputs can be validly refined by decreasing (increasing) its value for that input.

Before turning to an example, we explain how a set S can be increased or decreased in an ILP system.

Increasing or decreasing S. An aggregate condition like $max(S) \geq v$ can be validly refined by decreasing S, i.e., reducing the set to be aggregated. In ILP, this can be achieved by specializing the query Q used to generate the set.

An aggregate condition like $min(S) \geq v$ can be validly refined by increasing S. This can be obtained in ILP by generalizing the query that is used to generate the set, thus, by removing literals from it.

Example. We now illustrate the possible refinements with a small example. Suppose we have the following clause:

$$person(P, pos) \leftarrow max(\{A | child(P,C), age(C,A)\}) \geq 10.$$

The aggregate condition $F(S) \in I$ in this clause can be refined in three ways:

- decrease I (because the member function $m(R,I)$ is monotone in I)

  $$max(\{A | (child(P,C), age(C,A)\}) \geq 15.$$

- decrease $max(S)$ (because for $I = [v, \infty[$ the member function $m(R,I)$ is monotone in R). This can be achieved by
  - decreasing max (since the aggregate function $a(F,S)$ is monotone in F)

    $$avg(\{A | child(P,C), age(C,A)\}) \geq 10.$$
  - decreasing S (since $a(max,S)$ is monotone in S).

    $$max(\{A | child(P,C), age(C,A), male(C)\}) \geq 10.$$

3.4 The Refinement Cubes

We have presented three dimensions along which aggregate conditions can be refined: the aggregate function F, the query Q (or equivalently, the set S_Q, i.e., the set generated by Q), and the interval bound v. The whole set of hypotheses spanned by these three dimensions can be visualized in what we call a *refinement cube* (see Fig. 1). Every discrete point in the cube represents a hypothesis and can be constructed in a finite number of steps, starting from one aggregate condition. A chain of refinements in the cube will be called a *path*. For simplicity, we only consider refinements along one direction at a time. We are only interested in valid refinements, therefore we only allow *monotone paths* in the refinement cubes, i.e. paths that consist only of valid refinements.

For a given aggregate function class, refinement of aggregate conditions proceeds as follows. For every numeric attribute A, we look for a query Q that generates the set of values S_Q for each example. We take the smallest and largest aggregate function in the class (F_{small} and F_{large} respectively) and look for the smallest possible value returned by $F_{small}(S_Q)$ and the largest possible value returned by $F_{large}(S_Q)$ (V_{small} and V_{large} respectively). Then we construct two *start conditions*: $F_{large}(S_Q) \geq V_{small}$ and $F_{small}(S_Q) \leq V_{large}$. For each aggregate function class and each start condition there is a corresponding refinement cube that shows the allowed refinements.

The Refinement Cubes for the Generalized Averages. The generalized averages range from *min* to *max*. Possible thresholds for these functions range from the minimum to the maximum value in the dataset for the attribute under consideration. Hence, the two start conditions for this aggregate class are $max(S_Q) \geq min\_value$ and $min(S_Q) \leq max\_value$. The corresponding refinement cubes are shown in Fig. 1(a), with the start conditions indicated as a large dot. The arrows show the directions in which we can generate monotone paths starting from these aggregate conditions. Observe that when moving along the F-axis, the monotonicity properties of the aggregate function in S_Q change, so moving along the Q axis is only allowed in the top or bottom faces of the cube.

The Refinement Cubes for the Generalized Sums. Figure 1(b) shows the refinement cubes for the generalized sums. The V-axis ranges from the lowest value in the range of *max* (the minimal value for the numeric attribute) to the largest value in the range of *sum* (the maximum of the sum of the values for the attribute A, grouped by example, this value is called sum_value in the cube). The start conditions from which we can generate the whole cube are thus $sum(S_Q) \geq min\_value$ and $max(S_Q) \leq sum\_value$. The second start condition is anti-monotone in S_Q, and therefore positioned at the *specific* side of Q.

In this case moving along the F-axis does not change monotonicity (under the assumption that the generalized sums are only applied to sets of positive numbers), so the previous restriction does not apply here.

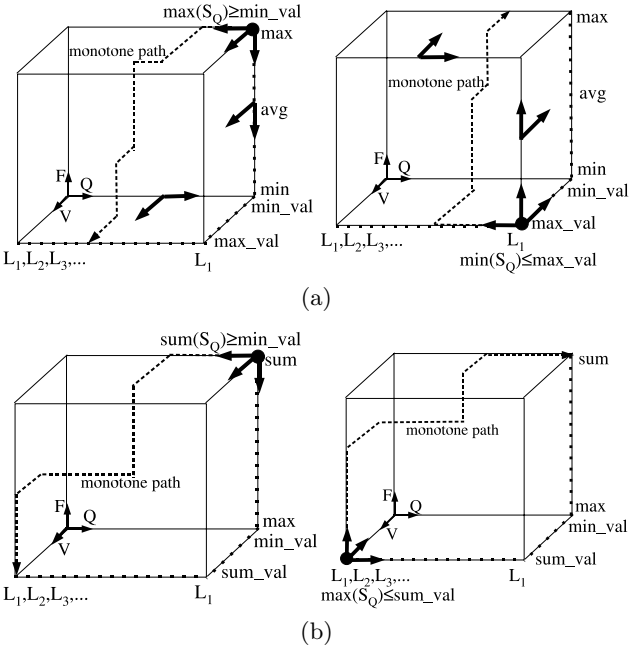

Fig. 1. (a) The refinement cubes for the generalized averages. (b) The refinement cubes for the generalized sums.

Remark that, when stretching the V-axis, the bottom face of the cubes can be connected to the top face of the cubes for the generalized averages, resulting in a combined refinement space.

The Refinement Cubes for the Generalized Counts. For the generalized counts the F-axis only contains the functions *count* and *count_distinct*. The V-axis ranges from 0 to the maximum size of the set generated by Q (*cnt_value*). The start conditions are $count(S_Q) \geq 0$ and $count\_dist(S_Q) \leq cnt\_value$. The monotone paths are the same as those for the generalized sums (see Fig. 1(b)).

Example. We now illustrate the use of the refinement cubes with an example. Consider the following start condition for the generalized sums:

$person(P, pos) \leftarrow sum(A, (child(P, C), age(C, A)), R), R \geq 10.$

Suppose we only use the functions *sum* and *max*, only use the threshold values 10 and 15, and only add a literal $male(C)$ to the query. Figure 2 schematically shows the refinements that are generated. Note that an aggregate condition can be obtained via more than one path, so in practice one has to take care to generate the conditions only once.

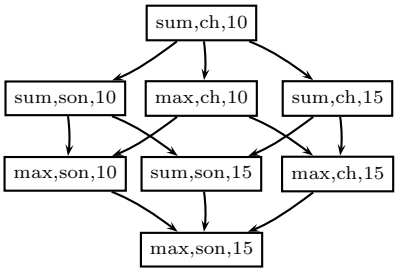

Fig. 2. Refinements generated for the start condition $max(\{A|child(P,C), age(C,A)\}) \geq 10$ by the cube for the generalized sums. In each node, the function F, query Q, and threshold value v are shown. The query is abbreviated: *ch* stands for $(child(P,C), age(C,A))$ and *son* means $(child(P,C), age(C,A), male(C))$.

Summary. Using the refinement cubes we obtain a search strategy that is

- *efficient*: only valid refinements are generated in each step, which allows to prune the aggregate search space,
- *complete*: every aggregate condition is reachable in a finite number of steps starting from the start conditions.

Remark that this efficiency can only be achieved by considering all three dimensions F, Q, and V together. A system that does not allow refinements along the F-axis (e.g., $max \rightarrow avg \rightarrow min$) can not obtain the aggregate conditions in a monotone (general-to-specific) way. For example, the condition $avg(\{A|child(P,C),\ age(C,A), male(C)\}) > 10$ can only be obtained via the monotone path $max(\{A|child(P,C), age(C,A)\}) > 10 \rightarrow max(\{A|child(P,C), age(C,A), male(C)\}) > 10 \rightarrow avg(\{A|child(P,C), age(C,A), male(C)\}) > 10$.

4 Application

In the previous section we have presented a general framework towards refining aggregate conditions. The framework can be beneficial for any relational learning system that learns aggregates and makes use of a general-to-specific ordering of the hypotheses to guide the search (e.g., decision tree learners, rule learners, frequent pattern miners,...). We now illustrate its usefulness by applying it to the relational decision tree learner Tilde [13] and showing that it yields a significant, though still modest, efficiency gain.

4.1 Tilde

Tilde is a relational decision tree learner. It learns trees with a divide and conquer algorithm similar to C4.5 [14]. The main point where it differs from the latter is that the tests to be considered at a node are Prolog queries. To split a

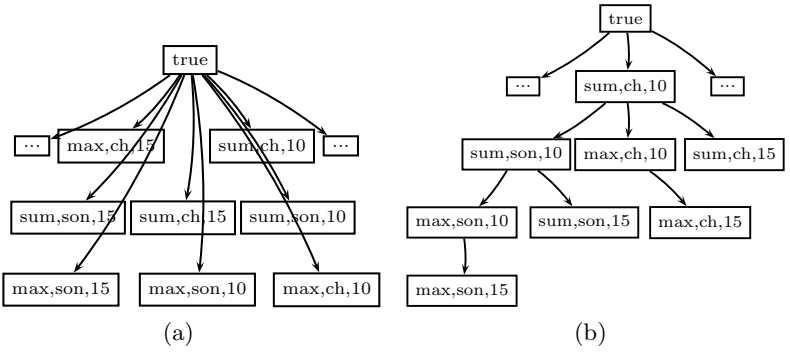

Fig. 3. (a) Original search space structure. (b) Optimized search space structure.

node, a refinement operator generates all allowed tests using a given hypothesis language specification. For each example corresponding to the node all tests are executed[2]. The test yielding the highest information gain is chosen.

Van Assche et al. [9] described how to add aggregate conditions to the hypothesis language of TILDE. Since the generality order on aggregate conditions was insufficiently understood, no structure was imposed on the aggregate search space, i.e., the tests in it were executed in random order. However, using the monotonicity properties discussed here, we know that whenever a test T fails for an example, none of the tests that can be obtained from T via monotone paths of the refinement cubes has to be executed against that example. Exploiting this knowledge would obviously result in an efficiency gain, while the same search space as before would be searched, and hence, the same tree would be obtained.

4.2 Experiments

We re-implemented TILDE's refinement operator in such a way that it structures the aggregate search space following the monotone paths of the refinement cubes[3]. Figure 3 shows how the search space for the example from Fig. 2 would be organized, both for the original and for the optimized refinement operator. If the test $sum(A, (child(P, C), age(C, A)), R), R \geq 10$ fails for an example, with the optimized method we could prune all its children in the search space, whereas in the original unstructured space, they would still all be tested.

In our experiments we compare the induction times obtained using the optimized refinement operator to those obtained when the original approach is used. In order to get the same set of tests at each node, we used the TILDE-LA setting [9] for the original refinement operator, where a lookahead of depth one is used

[2] This corresponds to the "examples in outerloop" approach (see [15]).

[3] In TILDE a query can only be refined by specializing it, not by dropping literals. Thus, in the context of the refinement cubes, refinements along the Q-axis can only take place in one direction.

Table 1. Runtime comparison of TILDE's optimized and original refinement operator

DATA SET	SEARCH	OPTIMIZED (SEC)		ORIGINAL (SEC)		SPEEDUP_RATIO	
	SP. SIZE	TOTAL	EXEC	TOTAL	EXEC	TOTAL	EXEC
MUTAGENESIS	53086	3792.54	2406.24	12079.97	10775.26	3.19	4.48
FINANCIAL	34900	4854.50	4717.08	12886.53	12751.98	2.65	2.70
DITERPENES	30533	12037.56	10518.78	16081.92	14535.39	1.34	1.38

in the aggregate query. For our comparison we used three datasets well-known in the ILP community: Mutagenesis [16], Financial [17], and Diterpenes [18].

For each of the experiments we used the aggregate functions *count*, *count distinct*, *max*, *avg*, and *min*. Table 1 gives an overview of the results we obtained[4]. The total runtime is reported, as well as the total time needed for executing the tests (summed over all nodes). Also the maximum size of the search space at a node in the tree is included. A first observation when looking at the table is that with the new method, a speedup factor of up to 4.5 is achieved for executing the tests. Second, since a large proportion of the total runtime goes into query execution, a significant overall runtime speedup is accomplished.

5 Conclusions and Future Work

The contributions of this paper are twofold. First, we have presented an in-depth study of the monotonicity of aggregate conditions, which is useful for refining models with aggregates. We have identified three dimensions along which monotonicity properties of an aggregate condition can be investigated: the aggregate function, the set to be aggregated, and the threshold value. This first dimension has never been explored before, but turns out to be crucial to obtain an efficient refinement strategy for aggregate conditions. Second, this derived strategy was applied to an existing relational decision tree learning system, illustrating the efficiency gain that can be achieved by using the results from this study.

An obvious direction for further work is to apply the framework to other relational systems. Another idea for future research is to search for aggregate function classes that contain other aggregate functions than the set we used.

Acknowledgements

Celine Vens is supported by the GOA 2003/8 project and by the Fund for Scientific Research (FWO) of Flanders. Jan Ramon and Hendrik Blockeel are postdoctoral fellows of the Fund for Scientific Research (FWO) of Flanders.

[4] Since accuracy, interpretability,... are unaffected by using this optimized refinement operator, they are not repeated here. We refer the reader to [9] for details.

References

1. Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained associations rules. In: SIGMOD International Conference on Management of Data. (1998) 13–24
2. Krogel, M.A., Wrobel, S.: Transformation-based learning using multi-relational aggregation. In: Proceedings of the 11th International Conference on Inductive Logic Programming. (2001) 142–155
3. Knobbe, A., de Haas, M., Siebes, A.: Propositionalisation and aggregates. In: Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery, Springer (2001) 277–288
4. Neville, J., Jensen, D., Friedland, L., Hay, M.: Learning relational probability trees. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2003)
5. Koller, D.: Probabilistic relational models. In: Proceedings of the 9th International Workshop on Inductive Logic Programming, Springer-Verlag (1999) 3–13
6. Perlich, C., Provost, F.: Aggregation-based feature invention and relational concept classes. In: Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM Press (2003) 167–176
7. Krogel, M.A., Wrobel, S.: Facets of aggregation approaches to propositionalization. In: Proceedings of the Work-in-Progress Track at the 13th International Conference on Inductive Logic Programming. (2003) 30–39
8. Knobbe, A., Siebes, A., Marseille, B.: Involving aggregate functions in multi-relational search. In: Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery, Springer-Verlag (2002) 287–298
9. Van Assche, A., Vens, C., Blockeel, H., Džeroski, S.: First order random forests: Learning relational classifiers with complex aggregates. Machine Learning, Special Issue on ILP (2006), to appear
10. Uwents, W., Blockeel, H.: Classifying relational data with neural networks. In: Proceedings of 15th International Conference on Inductive Logic Programming, Springer (2005) 384–396
11. Muggleton, S., ed.: Inductive Logic Programming. Academic Press (1992)
12. Plotkin, G.: A note on inductive generalization. Machine Intell. **5** (1969) 153–163
13. Blockeel, H., De Raedt, L.: Top-down induction of first order logical decision trees. Artificial Intelligence **101**(1-2) (1998) 285–297
14. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann series in Machine Learning. Morgan Kaufmann (1993)
15. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the efficiency of Inductive Logic Programming through the use of query packs. Journal of Artificial Intelligence Research **16** (2002) 135–166
16. Srinivasan, A., King, R., Bristol, D.: An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In: Proceedings of the 9th International Workshop on Inductive Logic Programming, Springer-Verlag (1999) 291–302
17. Berka, P.: Guide to the financial data set. In: The ECML/PKDD 2000 Discovery Challenge. (2000)
18. Džeroski, S., Schulze-Kremer, S., Heidtke, K.R., Siems, K., Wettschereck, D., Blockeel, H.: Diterpene structure elucidation from ^{13}C NMR spectra with inductive logic programming. Applied Artificial Intelligence **12**(5) (1998) 363–384