Incremental Construction of Complex Aggregates: Counting over a Secondary Table

Clément Charnay[1], Nicolas Lachiche[1], and Agnès Braud[1]

ICube, Université de Strasbourg, CNRS
300 Bd Sébastien Brant - CS 10413, F-67412 Illkirch Cedex
{charnay,nicolas.lachiche,agnes.braud}@unistra.fr

Abstract. In this paper, we discuss the integration of complex aggregates in the construction of logical decision trees. We review the use of complex aggregates in TILDE, which is based on an exhaustive search in the complex aggregate space. As opposed to such a combinatorial search, we introduce a hill-climbing approach to build complex aggregates incrementally.

1 Introduction and Context

Relational data mining deals with data represented by several tables. We focus on the typical setting where one table, the primary table, contains the target column, *i.e.* the attribute whose value is to be predicted, and has a one-to-many relationship with a secondary table. A possible way of handling such relationships is to use complex aggregates, *i.e.* to aggregate the objects of the secondary table which meet a given condition, using an aggregate function on the objects themselves (*count* function) or on a numerical attribute of the objects (*e.g. max*, *average* functions). For instance, we may want to classify molecules. Molecules have atoms. They can be represented as a table of molecules and a table of atoms, with a foreign key in the table of atoms indicating the molecule it belongs to. Then, the class of a molecule may depend on the comparison between the average of the charge of the carbon atoms of the molecule and some threshold value. This example also shows what a complex aggregate relies on: an aggregate function (here the *average*), a condition to select the objects to aggregate (here we select only the *carbon* atoms), the attribute to aggregate on (here the charge of the atoms), an operator and a threshold to make a comparison with the result of the aggregation.

Previous work showed that the expressivity of complex aggregates can be useful to solve problems such as urban blocks classification. [1,2] introduced complex aggregates in propositionalisation. But their use increases the size of the feature space too much, and they cannot be fully handled. This is the reason why we focus on introducing them in the learning step. To our knowledge, only one relational learner, TILDE [3], has implemented complex aggregates, but its exhaustive approach is not adapted to too complex problems. The main motivation for our work is to elaborate new heuristics to handle complex aggregates in

relational models. This article presents a logical decision tree learner which uses complex aggregates to deal with secondary tables, using a hill-climbing heuristic to build them incrementally. We introduce this heuristic in the context of logical decision trees, but it could be applied to other approaches. In this article, we focus on counting over a secondary table, *i.e.* the aggregate function considered will be the *count* function.

The rest of this paper is organized as follows. In Sect. 2, we review the use of complex aggregates in TILDE. In Sect. 3, we describe our heuristic to explore the complex aggregate space. Finally, in Sect. 4, we detail future work.

2 TILDE and Complex Aggregates

TILDE [3] is a first-order decision tree learner. It uses a top-down approach to choose, node by node, from root to leaves, the best refinement to split the training examples according to their class values, using gain ratio as a metric to guide the search. TILDE relies on a language bias: the user specifies the literals which can be added in the conjunction at a node. In this relational context, to deal with secondary tables, the initial version of TILDE introduces new variables from these secondary tables using an existential quantifier.

Then, TILDE has been extended [4] to allow the use of complex aggregates, and a heuristic has been developed to explore the search space [5]. This heuristic is based on the idea of a refinement cube, where refinement space for the aggregate condition, aggregate function and threshold are the dimensions of the cube. This cube is explored in a general-to-specific way, using monotone paths along the different dimensions: when a complex aggregate (a point in the refinement cube) is too specific (*i.e.* it fails for every training example), the search does not restart from this point.

However, the implementation does not allow more than two conjuncts in the aggregate query "due to memory problems" [6, p. 32], which limits the search space. Numerical attributes are handled by a comparison to a threshold in problems such as geographical ones. It is possible to discretize numerical attributes beforehand and to define predicates to make those comparisons between the values of the attributes and the thresholds given by the discretization. Nevertheless, enumerating all the combinations of thresholds over all the numerical attributes takes space in memory, and hence the approach is not tractable. To summarize, this combinatorial approach which handles complex aggregates in TILDE has limitations. We intend to overcome these limitations by not trying to explore the search space exhaustively, but by finding a heuristic to explore the search space and to build complex aggregates incrementally.

3 Incremental Construction of Complex Aggregates with Hill-Climbing

Our goal is to build a logical decision tree, like TILDE, which uses complex aggregates to deal with secondary tables. To explore the refinement cube of com-

plex aggregates, we choose to use a hill-climbing method. This section details the method. We use the general notation "$function(condition)\{operator\}threshold$" to refer to a complex aggregate.

3.1 Refinement of Complex Aggregates

When testing an aggregate, we make a refinement to find the best aggregate, and a hill-climbing is performed from a starting aggregate. In this paper, we limit ourselves to the *count* function to aggregate secondary tables. The starting conditions are detailed below. We then try to refine it with hill-climbing, allowing a non-strict climbing: if there is no strictly improving refinement, we allow picking a refinement with the same score as in the previous step, if it has not been visited before. To achieve that, we store all previous refinements chosen in the hill-climbing path. From the current aggregate, there are several ways to modify it:

- Firstly, given the examples, we compute the possible results of the aggregate function, which will serve as possible thresholds. To these values, we add one threshold depending on the operator: strictly lower than the other values if the operator is $\leq$ (so that the complex aggregate is true for none of the examples) and strictly higher if the operator is $\geq$. Such thresholds are chosen to be respectively MAX_DOUBLE and its opposite. The current threshold is then set to the closest possible threshold if it is lower than the minimum or higher than the maximum of the possible thresholds. For instance, if the current refinement to try is *"the count of atoms in the molecule is less than or equal to MAX_DOUBLE"* and, in the training set, there are between 13 and 42 atoms in a molecule, the threshold will be immediately set to 42.
- Then, we can refine the aggregate by increasing or decreasing the threshold (among the possible thresholds).
- Other possibilities are to remove a literal from the aggregate condition, or to add one.

The Starting Conditions Given this, if we refer to the maximum threshold as max, 2 starting conditions will be $count(true) \leq max$ and $count(true) \geq MAX\_DOUBLE$. From the point of view of the examples considered, they are opposite: if one succeeds for an example, the other will fail. The former is the most general (*i.e.* it succeds for every example), the latter the most specific. We then observe that refinements for one will have the same effect on information gain for the other (the final branch of the examples will be inverted between both), so on the training set, they are equivalent, and will be refined equivalently. In the end, we get two conditions $count(condition) \leq someThreshold$ and $count(condition) \geq nextThreshold$ which are opposite. To conclude on this point, considering both starting conditions is not necessary, since they will be refined following similar paths and will yield the same information gain after the hill-climbing process. Of course, the same reasoning applies for starting conditions $count(true) \leq -MAX\_DOUBLE$ and $count(true) \geq min$, this is the

reason why we consider only two starting conditions, arbitrarily with the operator $\geq$.

Moving in the Complex Aggregate Space We now discuss our method for refinement of the aggregate. If we add a literal to the aggregate condition, it will yield a specialization and less objects will be selected. The threshold range discussed above will not be the same, and the current threshold, associated to the previous, more general condition, will not be relevant since it may be too high. Hence, the refinement will yield a poor gain ratio and will not be chosen. For instance, in the training set, there are between 13 and 42 atoms in a molecule, but only between 5 and 20 carbon atoms. If the current aggregate states that "the count of atoms is less than or equal to 30", and we try to refine it to "the count of carbon atoms is less than or equal to 30", this last aggregate will be true for every example, yielding zero gain, and hence will not be chosen. Of course, the problem will be similar if we consider the other way, *i.e.* if we drop a literal from the aggregate condition, yielding a generalization.

To avoid modifying the aggregate condition without modifying the threshold, we do as follows. When modifying the aggregate condition, we consider the number of possible thresholds n_1 given by the current aggregate condition, and the number of thresholds n_2 given by the next (after modification) aggregate condition. We sort those two sets those thresholds in increasing order, such that the current threshold is in position c_1 with indices going from 0 to $n_1 - 1$, the next threshold chosen, in position c_2 between 0 and $n_2 - 1$ will be picked such that $\frac{c_2}{n_2-1}$ is closest to $\frac{c_1}{n_1-1}$. Mathematically: $c_2 = round(\frac{c_1 \cdot (n_2-1)}{n_1-1})$. Since we add a threshold to a list which already contains at least one element, there are always at least two possible thresholds and hence the case $n_1 = 1$ is not an issue.

3.2 Dealing with Empty Sets

We finally discuss a problem that will occur with aggregate functions other than *count*: the computation of a value for empty sets. Indeed, an aggregate condition might select no object, and aggregation over a numerical attribute is not possible in this case. For instance, how can the mean of the charge of the oxygen atoms in a molecule be computed when the molecule does not have any oxygen atom? Only the *count* function can deal in a natural way with empty sets, while another solution has to be chosen for numerical aggregate functions. Some possibilities to deal with this issue have been discussed in [7]:

- Fixing an arbitrary value as the result.
- Using a value depending on the aggregate condition, as close as possible to the values for examples for which the aggregate condition does not result in an empty set, or as far as possible.
- Failing the aggregate when the aggregate function cannot be applied.
- Discarding the aggregate from being chosen as a refinement if the aggregate function cannot be applied for at least one example.

In our opinion, the two first options are not easy to apply for every function. Indeed, for functions *min* or *max*, one can choose values as low or as high as possible so that the aggregate always succeeds or fails if the function cannot be applied directly. But for the *average* function, using a fixed value such as 0 is not relevant if the attribute can take both positive and negative values, neither is choosing positive or negative infinity. Moreover, the fact that the set to aggregate is empty can be meaningful, and by assigning a result to the aggregate function we lose this significance. The third option also gives to the empty set a meaning we do not necessarily want it to have: the failure of the aggregate should mean the inequality between the result of the aggregation and the threshold is wrong. However, this is not the meaning of the empty set. Actually, it is implied by the failure of the existential quantifier for the aggregate condition, *i.e.* $count(condition) \geq 1$ fails.

Then we see two ways to address the issue of empty sets: firstly to consider them as a third branch in our decision trees, since they do not correspond to a success or a failure of the inequality, they are a third possibility. However, this option breaks the binary structure of the tree, which is not necessarily a problem. Nevertheless, we present another option to preserve the binary tree structure: the principle is to create a node with the $count(condition) \geq 1$ aggregate before adding a node with an aggregate with a function which may not be applicable. In the left branch, we can then add the aggregate $function(condition) \geq threshold$ without the empty set problem, since we know from the parent node that *condition* will select at least one object. This adapts the three-branch idea to preserve the binary tree structure, using two nodes instead of one. An example is shown in Fig. 1, where the aggregate *"the average charge of the carbon atoms in the molecule is greater than or equal to 0.542"* is "protected" by the existential quantifier which tests the presence of at least one carbon atom in the molecule, *i.e.* the aggregate *"the count of carbon atoms in the molecule is greater than or equal to 1"*. If the latter succeeds, then the former can be evaluated because it is meaningful to compute the average value of a non-empty set. If the existential quantifier fails, then the average is not computed because it would be meaningless.

4 Conclusion and Future Work

The method described in Sect. 3 is implemented and will be evaluated. The next step in this work is to consider other aggregate functions, on numerical attributes of the secondary objects, and to allow the change of the aggregate function in the refining process of the aggregates, by taking advantage of their ordering as in [5]. Then, another step will be to allow recursivity in the aggregates, *i.e.* create complex aggregates which have complex aggregates in their aggregate condition, to use the whole database. However, this will inevitably raise new issues, since this will add other levels of refinements. A more complex problem will be to handle many-to-many relationships, since the complex aggregates can

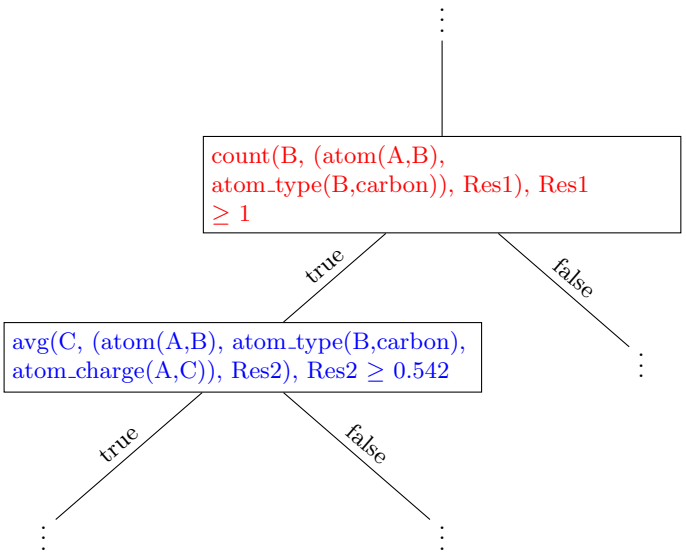

Fig. 1. Example of three-branch structure to deal with empty sets.

be formed both ways with such relationships, which can possibly lead to loops in the recursivity discussed above.

References

1. El Jelali, S., Braud, A., Lachiche, N.: Propositionalisation of continuous attributes beyond simple aggregation. In Riguzzi, F., Zelezný, F., eds.: ILP. Volume 7842 of Lecture Notes in Computer Science., Springer (2012) 32–44
2. Puissant, A., Lachiche, N., Skupinski, G., Braud, A., Perret, J., Mas, A.: Classification et évolution des tissus urbains à partir de données vectorielles. Revue Internationale de Géomatique **21**(4) (2011) 513–532
3. Blockeel, H., Raedt, L.D.: Top-down induction of first-order logical decision trees. Artif. Intell. **101**(1-2) (1998) 285–297
4. Assche, A.V., Vens, C., Blockeel, H., Dzeroski, S.: First order random forests: Learning relational classifiers with complex aggregates. Machine Learning **64**(1-3) (2006) 149–182
5. Vens, C., Ramon, J., Blockeel, H.: Refining aggregate conditions in relational learning. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: PKDD. Volume 4213 of Lecture Notes in Computer Science., Springer (2006) 383–394
6. Blockeel, H., Dehaspe, L., Ramon, J., Struyf, J., Assche, A.V., Vens, C., Fierens, D.: The ACE Data Mining System. (March 2009)
7. Vens, C.: Complex aggregates in relational learning. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science (March 2007) Blockeel, Hendrik (supervisor).