Feature Definition and Discovery in Probabilistic Relational Models

Eric Altendorf ERIC@CLEVERSET.COM
Bruce D'Ambrosio DAMBROSI@CLEVERSET.COM
CleverSet, Inc., 673 Jackson Avenue, Corvallis OR, 97330

Abstract

Feature expression in relational models can be viewed as the construction, for a given relational table, of valid path expressions and transformational operators which navigate the relational structure of the schema, propositionalizing native attributes from related tables via selection and aggregation. We present a language for expressing such features ("synthetic variables") and a method for efficiently searching over this language for definitions of relevant and interesting features for a probabilistic relational model.

1. Introduction

Relational and object models frequently make use of the notion of a *path expression*. A basic path expression, such as `Product.maker.name` , selects data from a table related to the root table of the original query. The use of path expressions in the context of probabilistic relational models (PRMs) has been discussed by Friedman et al. (1999) and Getoor et al. (2001).

Our first contribution has been the development of a path language with greater expressive power than those currently proposed. Some syntax and constructs were inspired by the PathLog language for OODBs developed by Frohn et al. (1994). However, the application to PRMs differs significantly and required both new features (such as arbitrary function chaining and data filtering), as well as a library of useful aggregators and operators. Our second contribution has been the development of automated tools to allow the modeler to make effective use of the richness of the language.

To motivate our discussion, consider the following examples of features we might wish to define (these examples are taken from a website user behavior model). Note that the first two are expressible in the language used in (Friedman et al., 1999; Getoor et al., 2001), while the rest are not.

- `Session.clicks.Count()` : For a given session, return the number of clicks that were in that session.
- `Session.clicks.url.Mode()` : For a session, return the most frequently requested URL.
- `Click.session.clicks.pagetype.Uniquify().Count()` : For a click, find the number of distinct page types visited in that session.
- `Click.session.clicks.Diff(.time,.prev.time).Mean()` : For a click, find its session and calculate the average time between clicks in that session.
- `Click.Diff(.session.clicks[.GT(.time, $src.time)][.Equals(.pagetype,"Checkout")].sequence_no, $src.sequence_no).Min()` : For a click, return the number of subsequent clicks in the session before the first request for a Checkout page—that is, the number of clicks until a purchase event.

2. Expression language

2.1. Grammar

Table 1. Synthetic variable abstract grammar

expr	:	rootelem{elem}+
elem	:	.field | .function | .this | variable | selector | constant | universal
rootelem	:	classname | variable | constant
selector	:	[expr]
function	:	funcname({expr}*)
constant	:	'[0-9]+{.[0-9]+}?' | "[^"]+"
variable	:	$ varname
universal	:	* classname

The grammar is defined in table 1. The basic construct in the language is an *expression*, which is a chain of elements. The first element is a *root element* which types the source datum, and subsequent elements define data mappings. Some elements contain and make use of subexpressions. Generally speaking, expressions may make sense only in specific contexts (for example, as

subexpressions). A *synthetic variable* is an expression which is well-defined given no special context other than the source datum on which it is to be evaluated.

A few minor points bear special mention. In the context of a subexpression, as a syntactic sugar we omit the classname element since it is implied by its context. Also, we name two special types of functions: *operators*, such as `Equals()`, `Diff()`, or `GT()` (greater than), which take two arguments, and *aggregators*, such as `Mean()` or `Count()`, which take no arguments and perform a many-to-one cardinality mapping (see 2.3). Finally, the `.this` element is a special no-op construct necessary in certain subexpression constructions.

2.2. Semantics

Expressions may be evaluated on an instance of the class associated with the root element. For example, `Session.clicks.Count()` may be evaluated on instances from the `Session` table, and `Session.clicks[ .Equals(.pagetype,.prev.pagetype)].Count()` contains three subexpressions: (i) `.Equals(.pagetype,.prev.pagetype)`, (ii) `.pagetype`, and (iii) `.prev.pagetype`, each of which may be evaluated on an instance from the `Click` table.

Evaluation of an expression proceeds left to right, each element accepting data from the previous element and producing data for the subsequent element. The value of the expression is the output of the last element in the chain.

2.2.1. Fields

The evaluation of a field element outputs the reference or primitive value(s) contained in the appropriate field on the incoming data object. If the incoming data is a singleton and the field is single-valued, the output is a singleton. Otherwise, the output is multivalued. If the incoming data is multivalued and the field is multivalued, the result will be a flattened set (the bag union of all field values on all instances from the incoming data); that is, we do not support nested collections.

2.2.2. Selectors and functions

The evaluation of a selector returns a subset of the incoming data—specifically, the collection containing each datum in the incoming data on which the provided subexpression evaluates to true.

The evaluation of a function returns a value based on the incoming data (and, if applicable, the values produced by evaluating the subexpressions on the incoming datum). For example, two-argument boolean operators such as `Equals()` or `GT()` take the incoming datum, evaluate each subexpression on that datum, and return a boolean value based on a comparison of those results.

Operators provide an implicit "map" behavior so that when applied to multivalued incoming data, they apply themselves to each datum in turn. Thus, one can construct variables such as `Click.session.clicks .Diff( .timestamp,.prev.timestamp).Mean()`.

2.2.3. Intra-expression variables

One seemingly expressive language construct which has turned out to be less useful than we predicted is arbitrary variable binding. The only important application we have found occurs when subexpressions refer to the original source datum on which the synthetic variable is being evaluated. We support this by implicitly binding the source datum to the `$src` variable, but (currently) do not offer a mechanism for arbitrary binding of variables (see 5.3).

2.2.4. Universal selection

In some datasets, we do not have explicit relations. Consider a spatiotemporal dataset, in which we would like to aggregate over spatially or temporally proximal events, but in which we lack fields (such as `adjacent`) to navigate the data. In this case, we must select *all* instances of a class, and then filter by our temporal or spatial conditions. We use the *universal selection* element, denoted by `*classname`, for this. For example, in the West Nile Virus domain (see 3.3.1), we have events indicating occurrences of the disease in humans and birds. To count the number of prior bird cases for a given human case, we can write: `HumanCase*BirdCase[ .GT($src.date,.date)].Count()` This functionality gives us the power to perform two-table joins on arbitrary conditions.

2.3. Typing

To facilitate type-checking, each element defines a required input and guaranteed output type. These types will depend on the relational schema, but not the data, meaning that expressions can be statically typechecked given the schema.

Datatypes are either reference (one type per table in the schema, with polymorphic subtyping allowed) or primitive[1]. Types also specify the cardinality of the data: singleton or multivalued (0 to n).

For an expression to be correctly typed, each element

[1]Currently string, numeric, or boolean, though we believe a type system based on measurement types such as nominal, ordinal, and ratio, might be more useful.

must accept the type of data produced by the preceding element. For example, field elements require input of the reference type (or subtype) on which that field is defined, aggregators require (possibly) multivalued inputs, single-input functions require singleton incoming data, and numeric aggregators like Mean() or Sum() require numeric input.

We require expressions to produce singleton data. This guarantees that synthetic variables can be evaluated to a single value, and that selectors and operators may evaluate their subexpressions to a single value.

2.4. Domain specific extensions

We have also built special purpose extensions within the grammar for particular problem domains. This capability is essential for doing useful applied modeling. Space constraints prohibit discussion, but some examples from spatiotemporal domains include: (i) functions for reifying and operating on temporal data, (ii) specialized selectors which allow rapid parameter adjustments for exploring scale effects, and (iii) density aggregators which normalize spatial data counts by the area of the sampled region.

3. Search

With a definition of the syntax and semantics of the language we can automatically enumerate synthetic variables. We view this problem as a search (for useful variables) in a very large search space (the space of all grammatically correct variables). An equivalent view is as a search over the space of possible database queries (Popescul & Ungar, 2003).

Our basic search is breadth-first, using a variant of the traditional cost ranking of path expressions by their *path length* (the number of field traversals they define).[2] Specifically, classname elements, which are in some sense artifacts of our grammar, cost nothing, and selectors and functions cost only what their their subexpressions cost.

3.1. Problem description

The space of possible synthetic variables without subexpressions is exponential in its length. (The base of the exponent is of course dependent on the relational schema and the aggregators enabled in the search.) Subexpressions introduce exponential branching—for instance, the number of selectors that may be applied at a given point will be exponential in the allowable size of subexpressions—and we therefore generally have superexponential overall growth.

As a very rough guide, in a simple schema, using no search optimizations or heuristics, search up to complexity (depth) 4 or 5 is generally reasonably tractable, but many interesting variables occur at complexities ranging from 8 to 12. In recent work with a major e-retailer, we have found valuable variables at complexities of 20 to 22. It is clear that such variables cannot be found by brute breadth-first enumeration, and so we introduce heuristics, synthetic variable filtering rules, and an interactive search process.

3.2. Synthetic variable filters

Besides restricting synthetic variables for grammatical and type correctness, we implement a number of filtering rules to prune the search space.

Field loop suppression: Traversals of one-to-many fields followed by their many-to-one inverse are generally useless, and so we suppress their generation. For example, while Product.maker.products makes sense (return all products produced by the same maker), Product.maker.products.maker is identical (up to repetition of data) as Product .maker . Loops of this nature may also span subexpression boundaries.

Repeated fields: The modeler may wish to limit chains of repeats of a transitive field, such as Click .prev , which could be arbitrarily extended to Click .prev.prev.prev. . . . The modeler can limit such repetitions on a per-field basis.

Field costs: The modeler may also wish to include certain fields more often or less often than others (due to an interest in a certain relationship, or to artifacts of the data's relational encoding). We allow the modeler to override the default cost of 1 on a per-field basis, to any positive value. Values greater than 1.0 shrink the search space, while values less than 1.0 increase it (but simultaneously cause variables including that field to appear at lesser search depths).

Field filtering: In some of our models data are related both spatially and temporally. However, our grammar, which is primarily navigational, does not provide a mechanism for multi-column joins. Therefore, we use a field traversal in combination with a filter, e.g.: Posbirds.geocell.posMosqPool [.Equals(.month,$src.month)] . By requiring that, say, temporal fields like .month are *only* used within selectors, we can generate expressions like the one above yet rule out ones like Posbirds.geocell

[2] We do offer other queue prioritizations—see 5.1.

.posMosqPool.month.posBirds . Although this may be a suboptimal solution, it has worked quite well in practice thus far.

Operator arguments: Arguments to a binary operator such as Equals() or GT() must be of the same type, but this does not prevent semantically meaningless combinations—product weight and maker's annual sales may both be numeric, but we shouldn't compare the two. We adopt the conservative rule that, for primitive types, comparisons are formed only between values from the same field on the same table.

Aggregator selection: It is very easy for the search routine to create vast numbers of variables merely by appending aggregators on the end of expressions, most of which are uninteresting. Therefore, we require that the modeler specifically enable aggregators in the search on a per-field basis.

Trivial domains: Although we focused on non-data-driven feature selection so we could operate in data-limited situations, we do implement a basic filtering rule for variables with trivial domains (i.e., constant). Variables may be constant because they aggregate large amounts of missing or constant data, or they may be provably axiomatic. While such variables may (occasionally) reveal something interesting about the domain or the data, they are not useful as random variables in a probabilistic model.

3.3. Empirical evaluation

The general quality of our results is encouraging. Recall (the proportion of desired variables which were found) appears to be quite good, and precision (proportion of variables returned which are interesting) is acceptable. For instance, at depth 9 in our West Nile Virus schema (see 3.3.1), we generate approximately 100 synthetic variables for the PosBirds table, compared to roughly 100,000 possible grammatically correct variables. The generated set includes all variables we had previously determined as essential to the model, and roughly 25% of the variables generated appear relevant to the model being constructed. Finally, the search is tractable; it takes between 30 seconds and several minutes on a desktop workstation.

We here present more detailed results of experiments we conducted to evaluate our pruning heuristics. Note that designing such experiments is difficult, because performance numbers and results vary highly. A slight change in the branching factor of the schema or the set of enabled aggregators could easily change the results for a given search by an order of magnitude. Additionally, there are a large number of free parameters, including (i) schema and set of enabled aggregators, (ii) amount of data populating the schema, (iii) depth of search, (iv) depth of subexpression search, and (v) filters applied.

3.3.1. Schemas used

We use two schemas, a weblog click-stream schema from an online retailer, and a geospatial epidemiological schema with data on the West Nile Virus in Maryland. In tables 2 and 3 we summarize the database tables, the number of instances (rows) in each, the numbers of each type of column, and the total number of aggregators enabled on various columns therein. Multivalued ("∗-val") reference columns are constructed as implicit inverses.

Table 2. Clickstream schema

Table	# of rows	Prim. cols	Ref. cols 1-val	Ref. cols ∗-val	# of agg.
Click	3789	7	4	0	0
Page	4823	8	0	0	0
Session	802	4	1	1	1
Visitor	563	8	0	0	0

Table 3. West Nile Virus schema

Table	# of rows	Prim. cols	Ref. cols 1-val	Ref. cols ∗-val	# of agg.
PosBirds	77	1	2	0	0
GeoCell	1248	0	0	7	1
Adjacent	9388	0	2	0	0
Month	13	1	1	4	0
PosMosqPl	10	1	2	0	1
NegMosqTr	73	1	2	0	1
Horse	5	1	2	0	1
Human	3	1	2	0	1
License	2116	1	1	0	0

3.3.2. Effectiveness of heuristics

Given a fixed schema, dataset, aggregator selection, and maximum subexpression complexity, we begin by searching with no heuristics, then enable them, one by one, and measure for each search depth: (i) the elapsed time, and (ii) the number of variables generated. The results are shown in figures 1, 2, 3, and 4. Missing data points indicate that that particular search either ran out of memory or time.[3] Almost all searches ran out of time or memory at depth 10. Space was generally a problem when operating with fewer heuristics, as the

[3]The memory limit was a 512MB Java heap size limit, and the time limit was 30 minutes. This of course is a research prototype, and much more efficient implementations are possible.

search created too many variables, while time was a problem when operating with more heuristics, as the search spent too long filtering (particularly in evaluating and removing variables with trivial domains).

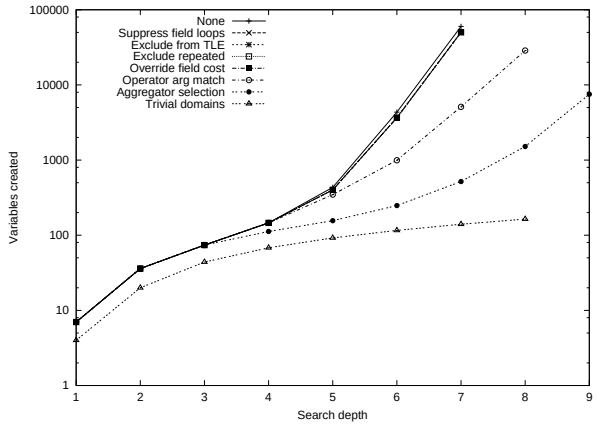

Figure 1. Number of variables created, clickstream schema

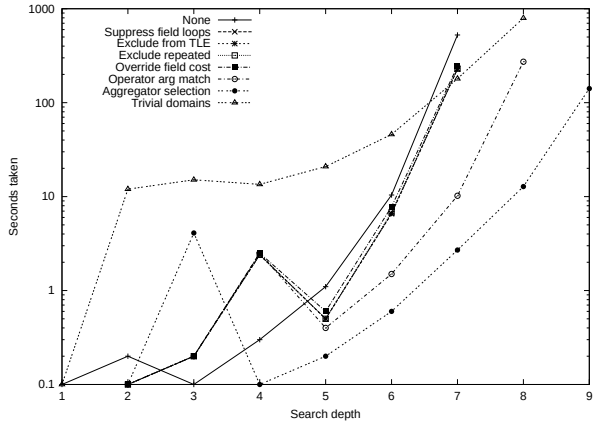

Figure 2. Time taken, clickstream schema

4. Interactive search process

Even by eliminating large regions of the search space the number of variables generated through the fully automatic search is generally too large for a PRM. We therefore designed the system with various features to make the feature selection process interactive.

4.1. Search results review

First, we separated feature search from model building. The modeler may perform an explicit synthetic variable search, and then review the results. Each synthetic variable in the results list may be evaluated on the loaded instance data, to produce either raw data or histograms, and can be deleted from the search results list or added to the PRM (for inclusion in later structure discovery).

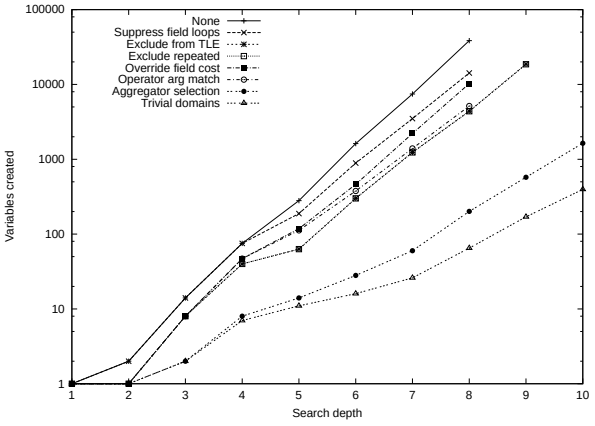

Figure 3. Number of variables created, WNV schema

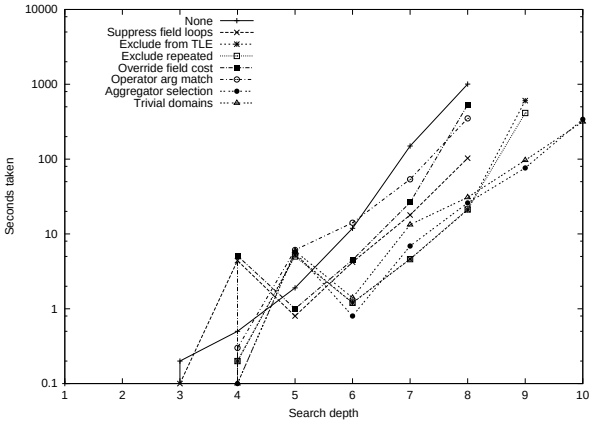

Figure 4. Time taken, WNV schema

4.2. Partial expressions

Second, we allow the creation of *partial expressions*. Partial expressions can be extended in a manual depth-first search, where possible next elements are automatically found and presented to the modeler for selection, or provided as a "seed" to the search algorithm, to automatically search for possible completions. In this way, with assistance from the modeler, the search algorithm can be used at arbitrary depths.

4.3. Complete manual specification

We also provide a parser which allows the modeler to input free-form synthetic variable definitions. We have found a number of interesting variables in a few domains which can only be created with the parser.

5. Future work

5.1. Search queue prioritizations

We search by producing continuations of partial expressions in a priority queue, normally prioritized by lowest expression complexity (as discussed). We also have two other scoring options, based on metrics calculated from complete expressions:[4]

Expression entropy: This lets us prioritize high-entropy variables and deprioritize nearly-constant variables. It is a generalization of the trivial domain filtering rule, which immediately removes from the search queue variables with zero entropy.

Log-likelihood: We prioritize by predictive power of the variable, measured by the log-likelihood of the data for modeler-selected "target" variables, given a model including the variable in question.

These techniques have not been empirically tested for performance, however, and deserve further study.

5.2. Extending the search space

There are many potentially interesting spaces over which we cannot automatically search, such as lengthy subexpressions, and values of constants passed as arguments to operators.

One approach is to develop new heuristics and search strategies. Another is to improve expression evaluation efficiency, since trivial domain testing (which requires evaluation of candidate synthetic variables) consumes the majority of time during search when the underlying dataset is large. Future research should address scalability issues, streamline evaluation, and/or develop statistically sound sampling techniques.

5.3. Intra-expression variables

Our original design called for general intra-expression variables which could be bound within an expression and used in other parts of the expression. We also considered allowing a variable bound multiple times to define an implicit join condition on the data, as it does in PathLog. Further research is needed to determine what, if any, expressive power such variables would add to the language (as we have not yet found a need for them).

[4]Partial expressions cannot be evaluated or scored on their own, so we prioritize them by an average of the scores of known complete expressions, weighted by a syntactic similarity metric.

5.4. Latent synthetic variables

One simplifying assumption we make in much of our work is that we do not need to do inference over data missing from a path expression. That is, we assume that the evaluation of a path expression is deterministic, or that path expressions are never latent variables of the model. Relaxing this assumption of course requires more advanced unrolling techniques to implement language elements as Bayes net fragments.

6. Conclusion

We have presented an expressive and efficiently evaluable language for defining derived attributes in relational models. Our experience has shown this language sufficiently expressive for a very wide range of real-world models. We have presented a set of heuristics for constraining searches over the space of possible expressions, and validated these heuristics empirically with performance tests, showing that they make the search space significantly more tractable. We have also discussed an interactive search process for going beyond the limits of fully automatic search. Finally, we discussed several important directions for valuable future research.

Note: parts of the described language and search capabilities are patent-pending.

References

Friedman, N., Getoor, L., Koller, D., & Pfeffer, A. (1999). Learning probabilistic relational models. *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)* (pp. 1300–1309). S.F.: Morgan Kaufmann Publishers.

Frohn, J., Lausen, G., & Uphoff, H. (1994). Access to objects by path expressions and rules. *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, Sep. 12-15, 1994, Santiago de Chile, Chile* (pp. 273–284). Morgan Kaufmann.

Getoor, L., Friedman, N., Koller, D., & Taskar, B. (2001). Learning probabilistic models of relational structure. *Proceedings of the 18th International Conference on Machine Learning* (pp. 170–177). Morgan Kaufmann, San Francisco, CA.

Popescul, A., & Ungar, L. H. (2003). Statistical relational learning for link prediction. *Proceedings of the IJCAI 2003 Workshop on Learning Statistical Models from Relational Data*.