A Link-Based Method for Propositionalization

Quang-Thang DINH, Matthieu EXBRAYAT, Christel VRAIN

LIFO, Bat. 3IA, Université d'Orléans
Rue Léonard de Vinci, B.P. 6759, F-45067 ORLEANS Cedex 2, France
{Thang.Dinh,Matthieu.Exbrayat,Christel.Vrain}@univ-orleans.fr
http://www.univ-orleans.fr/lifo/

Abstract. Propositionalization, a popular technique in Inductive Logic Programming, aims at converting a relational problem into an attribute-value one. An important facet of propositionalization consists in building a set of relevant features. To this end we propose a new method, based on a synthetic representation of the database, modeling the links between connected ground atoms. Comparing it to two state-of-the-art logic-based propositionalization techniques on three benchmarks, we show that our method leads to good results in supervised classification.

1 Introduction

Propositionalization is a popular technique in ILP, that aims at converting a relational problem into an attribute-value one [1–5]. Propositionalization usually is decomposed into two main steps: generating a set of useful attributes (features) starting from relational representations and then building an attribute-value table, which can be mono-instance (a single tuple for each example) or multi-instance (several tuples for an example). Traditional attribute-value algorithms can then be applied to solve the problem. Approaches for constructing automatically the new set of attributes (features) can be divided into two trends [6, 7]: methods based on logic *or* inspired from databases.

The first trend follows the ILP tradition which is logic-based. This trend, as far as we know, includes the first representative LINUS system [8] and its descendants, the latest being RSD [9], HiFi [4] and RELF [5]. For these systems, examples are mostly represented as first-order Herbrand interpretations and features are conjunctions of first-order function-free atoms. The search for features is based on a *template* (a set of ground atoms of which all arguments fall in exactly one of two categories: "input" or "output") or mode declarations (defining the predicates and assigning a *type* and *mode* to each of their arguments).

The second trend is inspired from databases and appeared later beginning with systems like Polka [2], RELAGGS [10] and RollUp [7]. Those systems build attributes, which summarize information stored in non-target tables by applying usual database aggregate functions such as count, min, max, etc.

In this paper, we propose a new method, called *Link-Based Propositionalization* or LBP, to build features for propositionalization from a set of ground atoms, without information on templates or mode declarations. The method was

initially designed to learn the structure of Markov logic networks [11], where it was used as a strategy to build a boolean table and to find dependent literals. The originality of the method is to build an abstract representation of sets of connected ground atoms, allowing thus to represent properties between objects.

LBP differs from the classical logic-based approaches both in the *semantic of the boolean table* and in the *search for features*. For example, the RELF system uses a block-wise technique to construct a set of tree-like conjunctive relational features while the others, like HIFI or RSD use the traditional level-wise approaches. The search in LBP does not rely on template or mode declarations, but on a synthetic representation of the dataset, namely the links of the chains, which allows to build features as well as to construct the boolean table based on the regularities of these chains. The notion of chain is related to relational path-finding [12] and relational cliché [13].

Our propositional method is presented in Section 2. We present related works in Section 3. Section 4 is devoted to experiments and finally, Section 5 concludes this paper.

2 Link-Based Propositionalization

Given as input a database DB and a query predicate Q, we present here a heuristic Link Based Propositionalization method (LBP) in order to transform relational information in data into an approximative representation in form of a boolean table. Once this boolean table has been learnt, it can be used for several tasks: looking for the most frequent patterns satisfied by instances of predicate Q, looking for the most discriminative patterns satisfied by positive examples of Q, or as input of a propositional learner for learning a model classifying positive from negative examples.

2.1 Preliminary notions

Let us recall here some basic notions of first order logic. We consider a function-free first order language composed of a set $\mathcal{P}$ of predicate symbols, a set C of constants and a set of variables. An *atom* is an expression $p(t_1, \ldots, t_k)$, where p is a predicate and t_i are either variables or constants. A *literal* is either a positive or a negative atom; it is called a *ground literal* when it contains no variable and a *variable literal* when it contains only variables. A *clause* is a disjunction of literals. Two ground atoms are *connected* if they share at least a constant (or argument).

A *variabilization* of a ground clause e, denoted by $var(e)$, is obtained by assigning a new variable to each constant and replacing all its occurrences accordingly.

The method that we propose is based on an abstract representation of sets of connected atoms, either ground atoms or variable atoms. This abstract representation is learned from sets of connected ground atoms and it is used to build sets of connected variable literals. Let us first introduce this representation.

2.2 An abstract representation

The idea underlying this method is to detect regularities in ground atoms: we expect that many chains of connected atoms are similar, and could thus be variabilized by a single chain. The similarity between chains of ground atoms is captured by the notion of links that we introduce in this paper and that models the relations between connected atoms.

Definition 1. *Let* g *and* s *be two ground literals (resp. two variable literals). A* link *between* g *and* s *is a list composed of the name of the predicates of* g *and* s *followed by the positions of the shared constants (resp. variables). It is written* link(g, s) = {G S g^0 s^0 / g^1 s^1 / ... } *where* G *and* S *are the predicate symbols of* g *and* s, $g^i \in [1, arity(g)]$, $s^i \in [1, arity(s)]$ *and the combinations / g^i s^i / mean that the constants respectively at position g^i in* g *and s^i in* s *are the same. If* g *and* s *do not share any constant then* link(g,s) *is empty.*

We are interested in representing the properties of sets of connected literals. In order to have a sequential representation of these properties, we consider only chains of literals defined as follows:

Definition 2. *A* chain *of ground literals (resp. variable literals) starting from a ground (resp. variable) literal g_1 is a list of ground (resp. variable) literals $\langle g_1, ..., g_k, ... \rangle$ such that $\forall i > 1$, $link(g_{i-1}, g_i)$ is not empty and every constant (resp. variable) shared by g_{i-1} and g_i is not shared by g_{j-1} and $g_j, 1 < j < i$. It is denoted by $chain(g_1) = \langle g_1, ..., g_k, ... \rangle$. The length of the chain is the number of atoms in it.*
The link of the chain $gc = \langle g_1, ..., g_k, ... \rangle$ *is the ordered list of links $link(g_i, g_{i+1})$, $i \geq 1$, denoted by $link(gc) = \langle link(g_1, g_2)/.../link(g_i, g_{i+1})/... \rangle$. The link of a chain composed of a single atom is the empty list. When a chain is composed of only two atoms, its link is the link between its two atoms. A chain of ground literals (resp. variable literals) is called, for short, a* ground chain *(resp. a* variable chain*).*

Let us notice that in this definition, it is only require that the variable shared by g_{i-1} and g_i is not used in previous links. But there may exist in g_{i-1} or in g_i some constants occurring in g_j, $j < i - 1$. Sometimes, it may be useful to know if a link has been obtained from a chain of ground atoms or from a chain of variable literals. In such situations, the term *link* is prefixed by *g-*, for expressing that the link has been obtained by a ground chain or by *v-*.

Definition 3. *A link $\langle g_1, ..., g_k \rangle$ is said to be a prefix of another link $\langle s_1, ..., s_n \rangle$, if $link(g_i, g_{i+1}) = link(s_i, s_{i+1})$, $\forall i, 1 \leq i < k$.*

Example 1. Let *DB1 = {P(a, b), Q(b, a), R(b, c), S(b), S(c)}* be a set of ground atoms. *P(a, b)* and *Q(b, a)* are connected by the two shared constants *a* and *b*. The constant *a* occurs respectively at position *1* of the ground atom *P(a, b)* and at position *2* of the ground atom *Q(b, a)*. Similarly, the constant *b* occurs

respectively at position *2* of the ground atom *P(a, b)* and at position *1* of the ground atom *Q(b, a)*. We have: *link(P(a, b), Q(b, a))* = $\{P\ Q\ 1\ 2\ /\ 2\ 1\}$.

A possible *chain* starting from the ground atom *P(a, b)* is $\langle$*P(a, b), R(b, c), S(c)*$\rangle$. Its link is $\langle\{P\ R\ 2\ 1\ \}\ /\ \{R\ S\ 2\ 1\}\rangle$. The link of the chain $\langle$*P(a, b), R(b, c)*$\rangle$ is $\langle\{P\ R\ 2\ 1\ \}\rangle$; it is a prefix of the previous link.

On the other hand, $\langle$*P(a, b), R(b, c), S(b)*$\rangle$ is not a chain as the constant b shared by *R(b, c)* and *S(b)* is already used to link *P(a, b)* and *R(b, c)*.

Definition 4. *A* variabilization of a link l *is a chain* c *of variable literals, so that* $link(c) = l$.

Let us for instance consider the link $\langle\{P\ Q\ 1\ 1\}\rangle$ where Q is a unary predicate. Then there is a single way, up to a renaming of variables, of variabilizing it, preserving the link, that is $P(A, B), Q(A)$. Nevertheless, given a link there may exist several ways of variabilizing it into a variable chain.

Let us now consider the link $\langle\{P\ R\ 1\ 2\}\ /\ \{R\ R\ 1\ 1\}\ \rangle$. This gives the scheme $P(slot_1, slot_2), R(slot_3, slot_4), R(slot_5, slot_6)$ with the constraints $slot_1 = slot_4$, $slot_1 \neq slot_3$, $slot_2 \neq slot_3$, $slot_2 \neq slot_4$ (thus satisfying the link $\{P\ R\ 1\ 2\}$) $slot_3 = slot_5$, $slot_3 \neq slot_6$, $slot_4 \neq slot_5$, $slot_4 \neq slot_6$ (for respecting the link $\{R\ R\ 1\ 1\}$) and the constraints $slot_3 \neq slot_4$ (for having a chain). Up to a renaming of variables, it can be variabilized into

$$P(X, Y), R(Z, X), R(Z, W) \text{ or into } P(X, Y), R(Z, X), R(Z, Y).$$

These two variabilizations correspond to two different strategies for filling the slots from left to right by variables, starting from the first atom composed of the first predicate and different variables as arguments. The first one consists in introducing a new variable each time there is no constraints on a slot of a predicate. The second one consists in trying to fill it with a variable that already exists, respecting the inequality constraints, thus leading to a more specific conjunction than the first one. This second strategy can still lead to several variabilizations, when several constants already introduced fulfill the constraints. The number of variabilizations can be reduced by using information on types of arguments when predicates are typed.

We define two strategies for variabilizing a link, and a third strategy for variabilizing a link, given the ground chain it comes from.

Definition 5. *Let* P *be the first predicate occurring in the link and let* n *be its arity. In both strategies, the first atom is* $P(X_1, \ldots, X_n)$. *Then slots are filled from left to right. For each slot with no equality constraints to fulfill:*

- *general variabilization: introduce a new variable*
- *specific variabilization: if possible, use a variable already introduced that fulfill all the inequalities constraints on this slot and the type of the argument.*
- *simple strategy: given a ground chain and its link, variabilize the ground chain, simply turning constants into variables.*

2.3 Creation of a set of features

Let us consider a target predicate Q and a training dataset *DB*. We aim at building a set of variable chains $\mathcal{F}$ linked to Q given *DB* such that for each

true ground atom A built with predicate Q in DB, and for each chain *chain(A)* starting from A, there exists a chain c in $\mathcal{F}$ such that $link(chain(A)) = link(c)$. It is reasonable to expect that many chains (starting from several ground atoms) are similar in the sense that their links are identical and could thus be variabilized by a single chain, with the same link.

The algorithm can be sketched as follows (in practice the length of the chains is limited by an integer k)

- for each ground atom A of the target predicate P,
  - find every chain starting from A
  - build the corresponding link and check whether it is a prefix of a link already built
  - if not, variabilize it.

In the current implementation, we have chosen the simple variabilization strategy, thus variabilizing the ground chain that has lead to the link under process. Moreover, we design two versions: in the first one (called LBP+), only positive ground atoms are considered, in the second one (called LBP-), positive and negative ground examples are considered.

Example 2. Let DB be a database composed of 14 ground atoms as follows: *advisedBy(bart, ada), student(bart), professor(ada), publication(t1,bart), publication(t2, bart), publication(t1, ada), publication(t2, ada), advisedBy(betty, alan), student(betty), professor(alan), publication(t3, betty), publication(t3, alan), publication(t3, andrew), professor(andrew).*

Figure 1 illustrates the production of chains of ground literals with the corresponding links and the resulting variable chains, using *advisedBy* as the target predicate, and bounding the length of chains to 4.

Starting from $advisedBy(bart, ada)$, several chains of ground literals can be built. For each chain, its link is built. For instance, the first chain built is {*advisedBy(bart, ada) student(bart)*}, leading to the link {*advisedBy student 1 1*}. This link is stored in the Set of Links and a first variable chain is created from this link.

The second chain {*advisedBy(bart, ada)*, *publication(t1, bart)*, *publication(t1, ada)*, *publication(t2,ada)*} leads to the link ⟨{*advisedBy publication 1 2*} / {*publication publication 1 1*}/ {*publication publication 2 2*}⟩. This link is not a prefix of the previous link, it is stored and a new variable chain is built from this link. Let us insist here on the fact that this chain depends on the variabilization strategy. The *general strategy* would lead to the chain {*advisedBy(A, B)*, *publication(C, A)*, *publication(C, D)*, *publication(E,D)*}. The *specific strategy* first introduces the new variable C as first argument of the first occurrence of *publication* (it cannot be equal to A or to B, otherwise it would have been written in the first link), leading to {*advisedBy(A, B)*, *publication(C, A)*}. When considering the second occurrence of publication, its first argument is given by the link. For its second argument, since no equality constraint is given on it, instead of introducing a new variable, it tries to use a previous variable: it cannot be A (it would have been given in the link), therefore it chooses B, leading to {*advisedBy(A, B)*, *publication(C, A)*, *publication(C,B)*}. Finally the third occurrence of *publication*

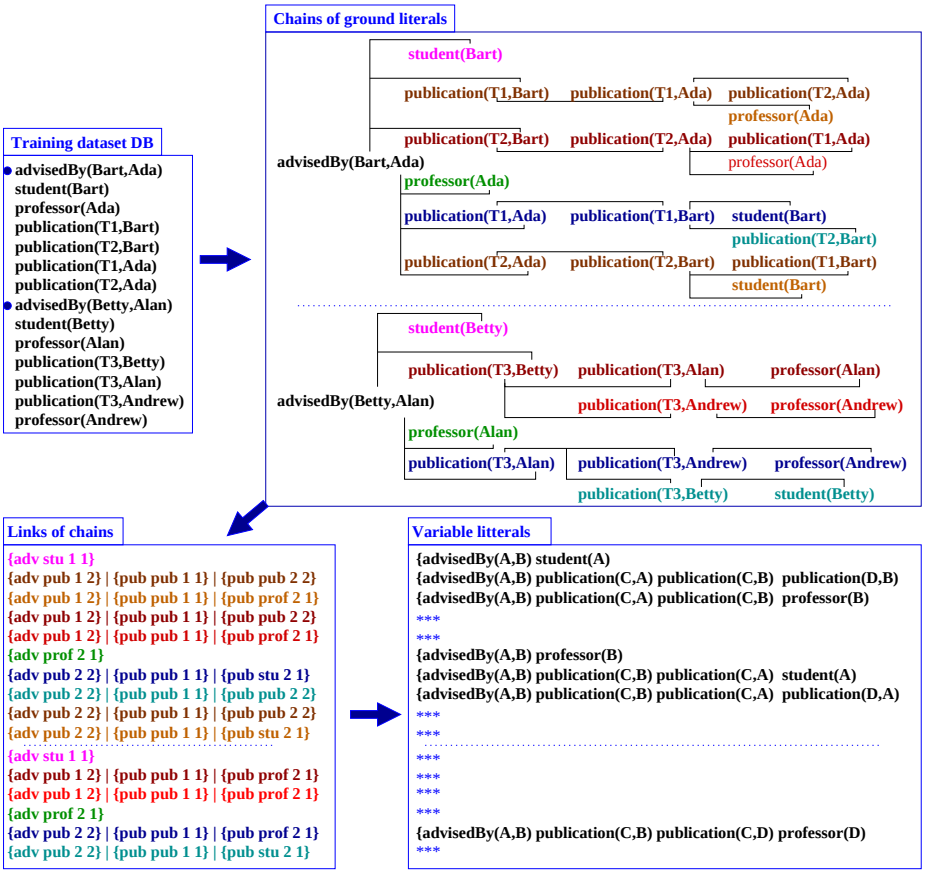

Fig. 1. The variabilization process using *chains* and *links* (length ≤ 4)

is introduced: its second argument is given by the link, no constraint is given on the first argument, no previous constant can be used (A and B do not have the same type and C cannot be used, because of the link. Thus we get {*advisedBy(A, B)*, *publication(C, A)*, *publication(C,B)*, *publication(D,B)*}. Since we know the ground chain that has lead to this link, the third strategy, called *simple strategy* variabilizes the ground chain, simply turning constants into variables. In this specific case, it leads to the same variable chain as the specific strategy. The simple strategy is the one we choose in the current implementation.

The third chain {*advisedBy(bart, ada)*, *publication(t1, bart)*, *publication(t1, ada)*, *professor(ada)*} leads to the link ⟨{*advisedBy publication 1 2*} / {*publication publication 1 1*}/ {*publication professor 2 1*}⟩. This link is not a prefix of the previous link, it is stored and a new variable chain is built from this link leading to {*advisedBy(A, B)*, *publication(C, A)*, *publication(C, B)*, *professor(B)*}.

The process goes on. For instance, the chain {*advisedBy(bart, ada)*, *publication(t2, bart)*, *publication (t2, ada)*, *publication(t1,ada)*} leads to the same link as the second chain, and it is not kept.

The three stars sign (***) displayed in Figure 1 means that there is no new variabilization for the corresponding chain. As can be seen, at the end, there are 16 ground chains starting from ground atoms built with *advisedBy* but only 7 different links and therefore 7 variable chains. Let us notice that the notion of chains allows to capture interesting relations, as for instance the relation between $advisedBy(A, B)$ and the fact that A and B have a common publication.

Let us notice that for improving the efficiency of the implementation, types information on predicates are used, as for instance *professor(person)*, *student (person)*, *advisedBy(person, person)*, *publication(title, person)*.

2.4 Creating a Set of Variable Literals and a Boolean Table

In classical approaches for propositionalization, the chains that have been built become features. Here, we intend to benefit from the structure of the chains. For instance, if we consider the variable chain {*advisedBy(A, B)*, *publication(C, B)*, *publication(C, D)*, *professor(D)*}, it can be split into 4 literals, with the requirement that given an instantiation of A and B (or a ground atom built on *advisedBy*), $professor(D)$ will be set to *true* if there exists an instantiation of C and of D such that *publication(C, B)*, *publication(C, D)*, *professor(D)* are true. Therefore $professor(D)$ equal to True means that the entire chain is true. On the other hand, $publication(C, D)$ may be true with only $publication(C, B)$ true.

Starting from the set of variable chains that has been built, we build a set of variable literals, renaming variables when necessary. In order to achieve this, we use a tree-structure representation of the chains. The idea is that for each variable literal, there exists a single chain linking this literal to the target literal. The process is divided into three steps: switch, sort and featurize.

The switch operation looks for sequences of two variable literals in two different chains that would be similar up to a permutation of these two literals, as for instance $publication(C, A), publication(C, B)$ in one chain and $publication(C, B)$, $publication(C, A)$ in the other one.

Then chains are sorted as follows: a chain c_1 precedes a chain c_2 if c_1 is shorter than c_2 or c_1 has the same length as c_2 and l_1 precedes l_2 where l_1 and l_2 are respectively the first literals in c_1 and c_2 that differ. The order relation between literals corresponds to the alphabetical order of the name of their predicates, or to the (alphabetical or numerical) order of the first pair of variables that differs if the two literasl are based on the same predicate. We must underline that such an order relation is only introduced in order to sort chains and that is should be given no further meaning.

A tree structure is then built, processing variable chains in turn. During this operation, variables can be renamed, thus allowing to distinguish features. A mapping table is used, linking the old name to the new one. More precisely, given a variable chain $l_1, \ldots, l_p$, we first look for a prefix already existing in the

tree, possibly using a variable renaming as given in the mapping table. Let us call i the last index of this prefix. Then we search whether any literal l_j, with $j > i$ already occurs in the tree, which means that if this literal was introduced "as is" in the tree, this later would contain two similar nodes at different places, and thus with different meanings. If so, then two cases are considered to overcome this potential problem:

- l_j contains only variables of the target predicate: the chain l_j, ..., l_p is forgotten, since there exists another shorter chain linking l_j with the target predicate.
- at least one of its variables is not a variable of the target predicate. This variable is renamed, introducing a new variable. The link between this new variable and the original one is kept in the mapping table.

Finally, the chain is introduced in the tree, renaming variables according to the mapping table. Let us notice that once renaming is done, it is possible that the common prefix detected in the original chain is no longer a common prefix (due to the renaming of the variable) and then a new branch is created as needed.

We can now see how switching and sorting lead to a valuable organization of chains. Due to the sequential introduction of chains in the tree, chains that share a common prefix become neighbors, sorted so that there introduction in the tree is likely to generate as less branches as possible.

Example 3. Let us consider again the database given in Example 2. The following links are built:

1. {*[advisedBy student/1 1]*}
2. {*[advisedBy professor/2 1]*}
3. {*[advisedBy publication/1 2], [publication publication/1 1], [publication professor/2 1]*}
4. {*[advisedBy publication/1 2], [publication publication/1 1], [publication publication/2 2]*}
5. {*[advisedBy publication/2 2], [publication publication/1 1], [publication student/2 1]*}
6. {*[advisedBy publication/2 2], [publication publication/1 1], [publication publication/2 2]*}
7. {*[advisedBy publication/2 2], [publication publication/1 1], [publication professor/2 1]* }

Then variable chains are built. (In the current implementation, variables are represented by an integer; here, we write them X_i to improve lisibility.)

1. {*advisedBy(X1,X2), student(X1)*}
2. {*advisedBy(X1,X2), professor(X2)*}
3. {*advisedBy(X1,X2), publication(X3, X1), publication(X3,X2), professor(X2)*}
4. {*advisedBy(X1,X2), publication(X3,X1), publication(X3,X2), publication(X4, X2)*}

5. {*advisedBy(X1,X2) , publication(X3,X2), publication(X3,X1), student(X1)*}
6. {*advisedBy(X1,X2), publication(X3,X2), publication(X3,X1), publication(X4, X1)*}
7. {*advisedBy(X1,X2), publication(X3,X2), publication(X3,X4), professor(X4)*}

The first step switches some consecutive literals, in order to favor common prefixes. It is applied on the 5th and 6th clauses, thus leading to:

5'. {*advisedBy(X1,X2), publication(X3,X1), publication(X3,X2) , student(X1)*}
6'. {*advisedBy(X1,X2), publication(X3,X1), publication(X3,X2) , publication(X4, X1)*}

In the second step, they are sorted:

1. {*advisedBy(X1,X2), professor(X2)*}
2. {*advisedBy(X1,X2), student(X1)*}
3. {*advisedBy(X1,X2), publication(X3, X1), publication(X3,X2), professor(X2)*}
4. {*advisedBy(X1,X2), publication(X3,X1), publication(X3,X2), publication(X4, X2)*}
5. {*advisedBy(X1,X2), publication(X3,X1), publication(X3,X2), publication(X4, X1)*}
6. {*advisedBy(X1,X2) , publication(X3,X1) , publication(X3,X2) , student(X1)*}
7. {*advisedBy(X1,X2), publication(X3,X2), publication(X3,X4), professor(X4)*}

In the context of this example, sorting as a limited impact, but in larger datasets it usually has a much more important influence on the organization of links. Then the tree is built. Chains 1, 2 and 3 have only $advisedBy(X1, X2)$ as a common prefix. Chain 3 has a literal, namely $professor(X_2)$, that contains only variables occurring in the head, this literal is then removed. Chains 4 and 5 have a prefix of length 3 common to chain 3 and then differs. In chain 6, the last literal is removed. Finally, in Chain 7, variable X_3 is renamed in X_5 in order to distinguish the two occurrences in different situations of $publication(X_3, X_2)$. This leads to the tree:

advisedBy(X1,X2)
— *professor(X2)*
— *student(X1)*
— *publication(X3,X1)*
——– *publication(X3,X2)*
———— *publication(X4,X1)*
———— *publication(X4,X2)*
— *publication(X5,X2)*
——– *publication(X5,X4)*
———— *professor(X4)*

Once the set of features is built, we transform information in the database into a boolean table BT, where each column corresponds to a variable literal and each row corresponds to a true/false ground atom of the target predicate. Let

us assume that data concerning a given ground atom q_r is stored in row r. Let us also assume that column c corresponds to a given variable literal vl_c. There exists a chain vc of variable literals starting from the head literal and containing vl_c. $BT[r][c] = true$ means that there exists a ground chain gc_r starting from the ground atom q_r that makes vc true (or such that $vc \subseteq var(gc_r)$ up to a renaming of variables). Given a ground instance q_r, filling its row by considering all the ground chains starting from q_r is too expensive because it has to involve an exhaustive search in the database. We overcome this obstacle by inversely considering the variable chains. Each of them is then used to guide the search in the database. This search can be performed much faster using information about the order of predicates and positions of shared constants between two consecutive predicates in that chain. The notion of links allows us to filter the variable chains thus reducing the search.

3 Related Works

As previously mentioned, propositionalization approaches can be classified into approaches based on logic and approaches inspired from databases. The approaches inspired from databases, like Polka [2], RELAGGS [10] and RollUp [7] build attributes which summarize information stored in non-target tables by applying usual database aggregate functions such as count, min, max, ...

These works are quite different from the ones based on logic, which consider examples as first-order Herbrand interpretations and features (attributes) as conjunctions of first-order function-free atoms. The search then is based on a template (a set of ground atoms of which all arguments fall in exactly one of three categories: input, output, or constant) or mode declarations (define the predicates and assign a type and mode to each argument of these predicates). Our method belongs to the second trend based on logic, and is compared to these approaches.

Logic-based approaches can be divided into two families : the first one as for instance [9, 5] starts from information on the predicates (usually the modes), build features and then uses the database to filter relevant ones, the second one as for instance [14] or our work starts from the database, build ground conjunctions of atoms and variabilize them to get features. Let us illustrate these two kinds of approaches.

In [5], the authors introduce the notion of templates and features. A *template* is a conjunction of ground atoms, the arguments of which are either defined as inputs (+) or as outputs(-). To be a template, an atom must have at most one input argument and there exists a partial irreflexive order on the terms occurring in it ($c < c'$ if c and c' occur in the same atom, c as input and c' as output. From a template, *conjunctions of connected variable literals* can be built, under the conditions that variables can be instantiated in such a way that the ground atoms belong to the template. A variable can be positive, when it has exactly one input occurrence and no output occurrence, negative when it has no input occurrence and exactly one output occurrence, or neutral when it has at least one

input occurrence and exactly one output occurrence. A *feature* is then a conjunction of connected variable literals where all variables are neutral. Intuitively it means, that each variable must be introduced as output of a single atom and can then be described by several atoms where it occurs as input. Features are built by aggregating blocks, where blocks are conjunctions of variable atoms, either containing exactly one positive variable (positive block) or containing exactly one negative variable (negative block).

In [14], conjunctions of literals are also built from the database. They use mode declarations and they distinguish two types of predicates: path predicates with at least one output argument and check predicates with only input arguments. The features (called properties in their approach) that they build must contain at least a check predicate. It means that "pure" relational features, as for instance expressing that two persons are linked if they share a common publication ({*advisedBy(A,B), publication(C,A), publication(C,B)*} cannot be built. The check predicates play an important role in the search strategy, since given a saturated ground clause, they start from the atoms built with a check literal and look for the path allowing to connect them to the head.

Our method differs from the classical logic-based approaches both in the *semantic of the boolean table* and in the *search for features*.

Meaning of the table: To form an attribute-value table, most methods define each propositional feature (column) corresponding to a variable literal (SINUS and DINUS [8] for example) or to a conjunction of several variable literals (the genetic method [15], RSD [9], HiFi [4] and RELF [5]). In LBP, each propositional feature (column) corresponds to a variable literal and *each row corresponds to a true/false ground atom of the query predicate*.

Searching: To construct features, most methods use syntactical constraints in the form of template or mode declarations for limiting the search space, then apply some techniques to calculate the truth values for each feature. For example, the RELF system uses a block-wise technique to construct a set of tree-like conjunctive relational features while the others, like HIFI or RSD use the traditional level-wise approaches. The search in LBP does not rely on template or mode declarations, but on a synthetic representation of the dataset, namely the links of the chains, which allows building features as well as constructing the boolean table based on the regularities of these chains. The notion of chain is related to relational path-finding [12] and relational cliché [13].

4 Experiments

4.1 Systems, Databases and Methodology

We propose to evaluate LBP according to classification accuracy, as traditionally used in propositionalization[4, 5, 7, 9]. Accuracy is relevant as it expresses the ability of LBP to produce discriminative features. More information is given in the form of the F1 score of both positive and negative groups.

We compared LBP to two state-of-the-art logic-based systems: RELF [5] and RSD [9]. For the comparison of logic-based and database-inspired methods, we refer to [6, 7, 16] for further reading. We performed experiments on three popular datasets:

- *IMDB* consists of a database on films (6 predicates, 302 constants, 1224 true/false ground atoms). We learned the predicate *workedUnder* (i.e. who worked under the direction of who).
- *UW-CSE* describes an academic department (15 predicates, 1323 constants, 2673 ground atoms). We learned the predicate *advisedBy* (i.e., who is the advisor of who).
- *CORA* consists of citations of computer science papers (10 predicates, 3079 constants, 70367 true/false ground atoms). We learned the predicate *sameBib* (i.e. do two citations refer to the same paper).

LBP has first been built over the Alchemy platform [1], since as written in the introduction, the idea of a linked-based representation of the database had first been introduced for learning Markov Logic networks [11]. The datasets have thus been used in their Alchemy form. Each set consists of 5 folds, which have been used for cross-validation. A new version of LBP, independent from Alchemy, has been built in Java and this is this new version that is used in the following experiments.

To evaluate the outputs of LBP, RELF and RSD, the set of features that have been produced and their corresponding boolean tables have then been given as inputs of a discriminative tool. The three systems produce output data in the Weka format, and we have chosen to test the discriminative power of the features on decision tree classifiers, using the WEKA [17] implementation of J48 and REPTree. We have chosen these two decision tree learners as they differ on the way trees are built: J48 implements the C4.5 algorithm while REPTree is a faster tree decision learner.

In most of cases, datasets are highly unbalanced. Given the closed world assumption, many negative examples could be generated. Therefore beside the pre-existing negative ground atoms of the target predicate, additional negative examples are generated randomly, to reach a rate of 4 negatives per positive. We mean, that based on a closed world assumption, we consider that all positive examples are explicitely given and that negative examples can be deduced from these latter. In the considered datasets, no or few negative examples are explicitely given. We thus keep these explicit negative examples and generate a subset of the implicit ones. This is empiric but it seems to be a fair way to get enough negative examples while not leading to too much overhead.

4.2 Dataset formats

The data input formats of the three systems we compared do differ and choices have to be made to encode the requirements of the systems. We briefly detail

[1] http://alchemy.cs.washington.edu/

how we proceeded to adapt them, starting from the Alchemy-like format. LBP requires only a database expressed by a set of ground atoms, positive and negative examples; type information can be given to improve the efficiency of the system. On the other hand, RELF and RSD need more information.

Alchemy-like data used in LBP consists of a ".db" file that contains the ground atoms. It usually comes with a ".mln" file that contains both the description of the predicates and some elements to help building a Markov logic network (which is out of the scope of this paper). The target predicate is defined at runtime. The description of the predicates can be very simple. In our case we only consider their name and arity, together with the type of arguments (e.g. *advisedBy(person, person)*). No mode or other additional declaration bias is used.

RELF learns by interpretation, which means that its input data consists of a set of independent world interpretations (i.e. a set of ground facts), which are annotated as valid or not. To comply to this data organization, we proceed as follows. Starting from each ground atom of the target predicate, we build an interpretation based on the saturation of this ground atom. We tag this interpretation as true of false based on the sign of the target atom. This latter is of course removed from the interpretation. In the case where the arity of the target predicate is higher that 1, as for instance *advisedBy*, we have to specify in the interpretation the constants occurring in the target atom. We thus introduce an additional predicate that is similar to the target one, but which is always true. Beside this data file, Relf needs an additional file that contains a template to guide its data exploration. Let us insist on the fact that this template has a high influence on the results.

RSD data input can take several forms. In the approach we use, input data consists of three files. First a ".b" knowledge base contains the mode declaration of the predicates that distinguishes between the target predicate (modeh) and the body predicates (modeb). The knowledge base also contains the positive ground atoms of the body predicates. Two other files are needed, a ".f" one and a ".n" one, that respectively contain the true and false ground atoms of the target predicate. The arity of the target predicate must be equal to 1. Thus, we have had to modify our datasets by introducing additional predicates: a new target predicate of arity 1, the variable of which is a new one, and *linking* predicates of arity 2 that link the new variable to the ones of the original target predicate. The new variable is set as an input one in the modes declaration.

We have used similar feature declaration biases for RSD and RELF. For LBP, we arbitrarily set the maximal length of considered g-chains (v-chains) to $k = 4$, in order to explore a rich while tractable search space. For RSD, due to the additional predicates we introduced, we set this maximum length to 6.

Beside dataset formats, we must also notice that LBP and RELF are able to take both learning and test datasets as input and produce the respective output files, while RSD is not. We thus adapted the tests as follows: with LBP and RELF we conducted a 5-fold cross validation based on our original folds, while with

RSD we merged all of the folds as a single input, thus getting a single output and letting Weka process with its built-in, 10-fold, cross validation.

4.3 Results

We present one table per dataset, containing for each system the global accuracy and the F1scores for positive examples (F+) and negative examples (F-). We used two versions of LBP, one that learns features based on the positive target atoms only (LBP+), and one that learns features based on both positive and negative target atoms(LBP-). Tables 1, 2 and 3 respectively correspond to the experiments with IMDB, UW-CSE and CORA. The presented results correspond to the average results of the 5- or 10-fold cross validation process. On each line the best average value if set in bold face.

		LBP+	LBP-	Relf	RSD
J48	Accuracy	**97.6**	92.4	92.8	84.6
	F+	**0.95**	0.85	0.86	0.90
	F-	**0.98**	0.946	0.95	0.69
REPTree	Accuracy	**97.6**	92.6	92.8	84.9
	F+	**0.95**	0.82	0.86	0.90
	F-	**0.98**	0.9	0.95	0.70

Table 1. Imdb experiments

		LBP+	LBP-	Relf	RSD
J48	Accuracy	90.4	**93.9**	91.1	85.8
	F+	0.79	0.86	0.81	**0.91**
	F-	0.94	**0.96**	0.94	0.71
REPTree	Accuracy	91.6	**94.5**	91.3	85.8
	F+	0.82	0.87	0.82	**0.91**
	F-	0.95	**0.96**	0.94	0.72

Table 2. Uw-cse experiments

We have got no results with RSD and REFL on Cora. More precisely, the systems terminate but provide no results. We can observe that in general the best accuracy is achieved with one of the two versions of LBP. Nevertheless, due to the fact that, depending on the dataset, either one or the other performs better, we cannot conclude that one of them is globally more performant from a statistical significance point of view.

		LBP+	LBP-	Relf	RSD
J48	Accuracy	87.9	86.8	-	-
	F+	0.76	0.74	-	-
	F-	0.92	0.91	-	-
REPTree	Accuracy	87.2	86.9	-	-
	F+	0.75	0.74	-	-
	F-	0.91	0.91	-	-

Table 3. Cora experiments

We also achieve the best F1Scores, except on UWCSE, where RSD performs better on the positive F1Score. Depending on the dataset, the decision tree learning algorithm might have some influence (UWCSE) or nearly no influence (IMDB) on the results at the average level. Nevertheless, even in this second case, differences might be noticed when considering some folds.

The important point is that these satisfying performances are obtained with a method that introduces no learning bias, except the types of variables, which is much lighter than the biases of REFL and RSD.

Considering time, LBP is the slowest system (about twice than the two other systems for UWCSE and IMDB). Implementation considerations might explain it partially, but on the large datasets, the fact that no declaration biases, such as modes, are available, makes the dataset exploration much longer.

Considering features, on UW-CSE, LBP+ produces c.a. 300 features, LBP- c.a. 550 features, RELF c.a. 130 features and RSD c.a. 300 features. On IMDB, LBP+ produces c.a. 30 features, LBP- c.a. 40 features, RELF c.a. 10 features and RSD c.a. 100 features. The fact that we produce much more features than RELF can be explained by at least two reasons. First, we produce features that consist of a single variable literal. We thus have several features when other systems produce a single conjunction. Second, due to the tree structure of our graph of features, we have a lot of features that are set and kept "just to" materialize a path to a leaf feature. Nevertheless, we surprisingly produce less features than RSD.

Based on these results, we can conclude that our system is competitive to the state-of-the-art propositional systems on these three benchmark datasets.

5 Conclusion and Future Work

In this paper, we introduce a linked-based representation of the database allowing to capture relational structures in the database, and we give a first way of integrating it in a propositional learner. Our main contribution is mainly on this linked-based representation allowing to learn features with nearly no information (except types of predicates). Another original contribution is the idea of splitting features into literals, relying on the fact that they form a chain.

Further works can be done on this representation. In the system that we have developed, learned features are split into ground atoms. Such features could also be used as such as in traditional propositional learners.

Although the system was designed for avoiding the user to give biases, modes could easily be added, thus allowing to reduce the number of links. On the other hand, most logical-based propositional learners need information: at least type and mode declarations for predicates, more sophisticated information, as for instance templates, which allows to reduce the search space. Giving such templates is not so easy. Our linked-based representation could perhaps be used as a preliminary step to learn templates.

References

1. Alphonse, É., Rouveirol, C.: Selective propositionalization for relational learning. In: PKDD'99. Volume 1704 of LNCS, Springer (1999) 271–276
2. Knobbe, A.J., de Haas, M., Siebes, A.: Propositionalisation and aggregates. In: PKDD'01. Volume 2168 of LNCS, Springer (2001) 277–288
3. De Raedt, L.: Logical and Relational Learning. Springer (2008)
4. Kuželka, O., Železný, F.: Hifi: Tractable propositionalization through hierarchical feature construction. In: Late Breaking Papers, ILP'08. (2008)
5. Kuželka, O., Železný, F.: Block-wise construction of tree-like relational features with monotone reducibility and redundancy. Mach. Learn. **83**(2) (2011) 163–192
6. Krogel, M.A., Rawles, S., Železný, F., Flach, P.A., Lavrac, N., Wrobel, S.: Comparative evaluation of approaches to propositionalization. In: ILP'03. Volume 2835 of LNCS, Springer (2003) 197–214
7. Lesbegueries, J., Lachiche, N., Braud, A.: A propositionalisation that preserves more continuous attribute domains. In: ILP'09. (2009)
8. Lavrac, N., Dzeroski, S.: Inductive Logic Programming: Techniques and Applications. Ellis Horwood (1994)
9. Lavrac, N., Zelezný, F., Flach, P.A.: Rsd: Relational subgroup discovery through first-order feature construction. In: ILP'02. Volume 2583 of LNCS, Springer (2002) 149–165
10. Krogel, M.A., Wrobel, S.: Transformation-based learning using multirelational aggregation. In: ILP'01. Volume 2157 of ILP, Springer (2001) 142–155
11. Dinh, Q.T., Exbrayat, M., Vrain, C.: Discriminative markov logic network structure learning based on propositionalization and chi^2-test. In: ADMA'10. Volume 6440 of LNCS, Springer (2010) 24–35
12. Richards, B.L., Mooney, R.J.: Learning relations by pathfinding. In: AAAI'92, AAAI Press / The MIT Press (1992) 50–55
13. Silverstein, G., Pazzani, M.J.: Relational clichés: Constraining induction during relational learning. In: ML'91, Morgan Kaufmann (1991) 203–207
14. Motoyama, J., Urazawa, S., Nakano, T., Inuzuka, N.: A mining algorithm using property items extracted from sampled examples. In: ILP. (2006) 335–350
15. Braud, A., Vrain, C.: A genetic algorithm for propositionalization. In: ILP. (2001) 27–40
16. Kuzelka, O., Zelezný, F.: Block-wise construction of acyclic relational features with monotone irreducibility and relevancy properties. In: ICML'09, ACM (2009) 72
17. Machine Learning Group at University of Waikato: Data mining software in java. `http://www.cs.waikato.ac.nz/ml/weka/`