Learning Linear Constraints in Inductive Logic Programming

Lionel Martin and Christel Vrain

LIFO - Université d'Orléans - BP 6759
45067 Orléans Cedex 2 - France
email: {martin,cv}@lifo.univ-orleans.fr

Abstract. In this paper, we present a system, called ICC, that learns constrained logic programs containing function symbols. The particularity of our approach is to consider, as in the field of Constraint Logic Programming, a specific computation domain and to handle terms by taking into account their values in this domain. Nevertheless, an earlier version of our system was only able to learn constraints $X_i = t$, where X_i is a variable and t is a term. We propose here a method for learning linear constraints. It has already been a lot studied in the field of Statistical Learning Theory and for learning Oblic Decision Trees. As far as we know, the originality of our approach is to rely on a Linear Programming solver. Moreover, integrating it in ICC enables to learn non linear constraints.

1 Introduction

This paper is mainly devoted to the problem of handling numeric data in Inductive Logic Programming. This involves several points: defining a formal framework for studying this problem, learning numeric relations, introducing functional terms. From the semantic point of view, a consensus has emerged about the semantics of definite logic programs in terms of Herbrand interpretations, whereas several semantics have been defined for normal logic programs that can be based either upon the classical two-valued logics or upon multi-valued logics. Nevertheless, pure logic programs do not enable to express numeric expressions. In Prolog, meta-predicates, as for instance the primitive *is a*, have been defined to deal with numeric values, but the semantics of such programs can no longer be studied in terms of Herbrand interpretations. To deal with this problem, a new field, called Constraint Logic Programming, has rapidly grown: it enables to express constraints that are interpreted in a specific domain of computation, as for instance, the set of integers, or the reals.

In this paper, we propose a new approach for learning logic programs containing function symbols. Instead of relying on the syntactic form of terms, we propose to consider a domain of computation and to build new terms based on their values in this domain and on the interest of these values for discriminating positive and negative examples. In an earlier version [6], the prototype, called ICC, that we have developed in the framework of Constraint Logic Programming, was limited to constraints $X_i = t$, where X_i is a variable and t is a term in which X_i does not occur. The main contribution of this paper is to propose a method, based on Linear Programming techniques, to learn relevant linear constraints and to integrate it in the system ICC, in order to learn non linear constraints.

This paper is organized as follows. Section 2 recalls basic definitions about Constraint Logic Programming. In Section 3, the system ICC is described. In Section 4, a method to learn linear inequalities is proposed as well as the way it is integrated in ICC.

2 Basic definitions

2.1 Syntax

We briefly recall some basic notions about constraint logic programming, that can be found in [5, 2]. We consider:

- an infinite set of variables $\mathcal{V}$,
- a set, denoted by Σ, of function symbols,
- a set, denoted by Π_C, of constraint predicate symbols, containing at least the predicate $=$,
- a set, denoted by Π_P, of predicate symbols definable by a program.

A *term* over $\mathcal{V}$ and Σ is inductively defined as follows: a variable v of $\mathcal{V}$ is a term and, if f is a function symbol of Σ, if n is the arity of f, $n \geq 0$ and if $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

A *primitive constraint* has the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of Π_C and $t_1, \ldots, t_n$ are terms.

A *constraint* is a first-order formula built with primitive constraints. In the remaining of the paper, we consider only the class $\mathcal{L}_\Sigma$ of constraints defined as the smallest set of constraints that contains all primitive constraints and is closed under variable renaming, conjunction and existential quantification.

An *atom* has the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of Π_P and $t_1, \ldots, t_n$ are terms.

A *constrained atom* is a pair (c, p), where c is a constraint and p is an atom.

A *constrained clause* is an expression $a \leftarrow c \square b_1, \ldots, b_n$, where c is a constraint and $b_1, \ldots b_n$ are atoms.

A *CLP program*, also called a *constrained program*, is a collection of constrained clauses.

Example 1. The following CLP program:

$facto(X, Y) \leftarrow X = 0, Y = s(0).$

$facto(X, Y) \leftarrow X = s(Z), Y = T * X \square facto(Z, T).$

defines *factorial* ($\Sigma = \{0, s, *\}$, $\Pi_C = \{=\}$ and $\Pi_P = \{facto\}$).

The variable X is *linked* in the constrained clause $A_0 \leftarrow c_1, \ldots, c_m \square A_1, \ldots, A_n$, if X occurs in A_0 or, if there is a primitive constraint c_i or an atom A_j that contains the variables X and Y such that Y is linked in C. A constrained clause C is *linked* if all its variables are linked.

2.2 The constraint domain and its semantics

In the field of Constraint Logic Programming, we usually consider a specific constraint domain over which computation is performed. A (Σ, Π_C)*-structure* $\mathcal{D}$ is composed of a non-empty set D, an assignment of a function $f_{\mathcal{D}} : D^n \rightarrow D$, for each $f \in \Sigma$, and an assignment of a function $p_{\mathcal{D}} : D^n \rightarrow \{True, False\}$, for each $p \in \Pi_C$.

Example 2. In Example 1, a (Σ, Π_C)-structure can be defined on the domain D of positive integers. It interprets the function 0 by the positive integer 0, the function s as the usual function *successor* on the set of positive integers, the function $*$ as the usual multiplication on the set of positive integers.

If $\mathcal{D}$ is a (Σ, Π_C)-structure, a $\mathcal{D}$*-atom* has the form $p(d_1, \ldots, d_n)$, where $p \in \Pi_P$ and $d_i \in D$, $1 \leq 1 \leq n$. The set of all $\mathcal{D}$-atoms is called the $\mathcal{D}$*-base.*

Let $\mathcal{D}$ be a (Σ, Π_C)-structure and let t be a ground term. The interpretation of the ground term t, w.r.t. $\mathcal{D}$, denoted by $I_{\mathcal{D}}(t)$, is defined as follows: if $f \in \Sigma$ and if $t_1, \ldots, t_n$ are terms, then $I_{\mathcal{D}}(f(t_1, \ldots, t_n)) = f_{\mathcal{D}}(I_{\mathcal{D}}(t_1), \ldots, I_{\mathcal{D}}(t_n))$,

In the following, $Const(F)$ denotes the set of constants of D appearing in the expression F and $Term_{\Sigma}(D')$ (where D' is a set of constants of D) denotes the set of terms built with constants of D' and function symbols of Σ.

If $\mathcal{D}$ is a (Σ, Π_C)-structure, a *valuation* v is a function: $\mathcal{V} \rightarrow D$.
Let v be a valuation, $v = \{X_1/d_1, \ldots, X_n/d_n\}$ where $d_1, \ldots, d_n \in D$. The *set of inverse valuations* of v, denoted by $V^{-1}(v)$, is defined by: $v^{-1} \in V^{-1}(v)$ iff $v^{-1} = \{d_1/X_{i_1}, \ldots, d_n/X_{i_n}\}$ with $v(X_{i_j}) = d_j$.
Let $A = p(d_1, \ldots, d_n)$ be a $\mathcal{D}$-atom and let v be the valuation $\{X_1/d_1, \ldots, X_n/d_n\}$. If $v^{-1} \in V^{-1}(v)$, $v^{-1} = \{d_1/X_{i_1}, \ldots, d_n/X_{i_n}\}$, then $v^{-1}(A)$ denotes the atoms $p(X_{i_1}, \ldots, X_{i_n})$.

Let $\mathcal{D}$ be a (Σ, Π_C)-structure and let T be a set of terms. An *inverse interpretation* of a constant d w.r.t T is the subset of T composed of the terms t that satisfy $I_{\mathcal{D}}(t) = d$. This set is denoted by $I_T^{-1}(d)$.

3 The system ICC

The learning task of ICC is specified by:

1. a set Σ of function symbols and a set Π_C of constraint predicates,
2. a (Σ, Π_C)-structure $\mathcal{D}$,
3. a set BASE of basic predicates defined by a set, denoted by BK^+, of $\mathcal{D}$-atoms,
4. a set TARG of target predicates, specified by two sets of $\mathcal{D}$-atoms: E^+ and E^- with $\text{E}^+ \cap \text{E}^- = \emptyset$. The set $\text{E} = \text{E}^+ \cup \neg\text{E}^-$ represents the intended interpretation and in the following, Π_P denotes the set BASE $\cup$ TARG.

A $\mathcal{D}$-atom e is $\mathcal{D}$-covered with respect to $E^+ \cup BK^+$ by a constrained clause C $= A_0 \leftarrow c \,\Box\, A_1, \ldots, A_n$ iff there exists a valuation v which satisfies c and such that $v(A_0) = e$ and $v(A_i) \in E^+ \cup BK^+$ for $i = 1 \ldots n$. Such a valuation is called a *covering valuation for e.*

The aim of ICC is to find a constrained program, built over Σ, Π_C, and Π_P which $\mathcal{D}$-covers the positive examples and which $\mathcal{D}$-covers no negative ones.
The method used to build linked constrained clauses is a classical one, that consists in iteratively adding to the body of the clause, either a constrained atom or a constraint until no negative example is $\mathcal{D}$-covered. We recall here the way relevant constraints are built. A similar method has been developed for building relevant constrained atoms. For more details, see [6].

Since a clause must $\mathcal{D}$-covers at least a $\mathcal{D}$-uncovered positive example, the first step of the algorithm is to choose (randomly) a $\mathcal{D}$-uncovered positive example $e \in E^+$ and to build a clause which $\mathcal{D}$-covers e and as many $\mathcal{D}$-uncovered positive examples as possible. The positive example e enables us to build a set of relevant constraints and a set of relevant constrained atoms, and then a classical entropy measure [10] enables to choose the best constraint or the best constrained atoms. Therefore, let $e \in E^+$ be a $\mathcal{D}$-uncovered positive example, let C be a clause that $\mathcal{D}$-covers e and some negative examples, and let v be a covering valuation for e. The set of primitive constraints, denoted by *Constr*, that can be added to C in order to refine it, is computed as follows:

Algorithm 1. Main Algorithm of ICC.

```
1- Compute the set $T = Term_\Sigma(Const(v))$.
2- Compute $D' = \{I_\mathcal{D}(t) | t \in T\}$.
3- Build $PC = \{p_i(d_{i_1}, \ldots, d_{i_n})\} | p_i \in \Pi_C, d_{i_j} \in D', p_i(d_{i_1}, \ldots, d_{i_n})$ true in $\mathcal{D}\}$.
4- Compute $I_T^{-1}(PC) = \{I_T^{-1}(cc) | cc \in PC\}$
5- Remove trivial constraints from $I_T^{-1}(PC)$
6- Compute $Constr = \bigcup_{v^{-1} \in V^{-1}(v)} v^{-1}(I_T^{-1}(PC))$
```

Example 3. Biases are introduced in ICC to reduce the set T, which have enabled ICC to learn the following constrained logic programs, respectively defining *factorial* and *member.*

Learned program	$\lvert E^+ \rvert$	$\lvert E^- \rvert$
$facto(X, Y) \leftarrow X = 0, Y = succ(0).$ $facto(X, Y) \leftarrow Z = pred(X), T = div(Y, X) \,\Box\, facto(Z, T).$	7	6
$member(X, Y) \leftarrow X = head(Y).$ $member(X, Y) \leftarrow Z = X, T = body(Y) \,\Box\, member(Z, T).$	12	6

As has been mentioned in the introductory section, the constraints learned by ICC have been limited to expressions $X_i = t$, where X_i denotes a variable and t a term. In Section 4, we present a method for learning linear inequalities and in Section 5, we explain how it is integrated in ICC.

4 Learning linear constraints

Let us suppose that we have built the clause $b_0 \leftarrow c \,\square\, b_1, \ldots b_n$ and let us call $X_1, \ldots, X_n$ the numeric variables that have been introduced in this clause. Our goal is to find a linear constraint:

$$c' : a_0 + a_1 * X_1 + \ldots + a_n * X_n \leq 0$$

such that the clause $b_0 \leftarrow c, c' \,\square\, b_1, \ldots b_n$ $\mathcal{D}$-covers a maximum of positive examples and a minimum of negative examples. The problem $\mathcal{P}$ that must be solved can be, more generally, stated as follows:

Let $X_1, \ldots, X_n$ be n numeric variables. Let $[v_i, lab_i]_{1 \leq i \leq m}$ be m examples where v_i denotes a valuation: $\{X_1, \ldots, X_n\} \rightarrow D$ and $lab_i \in \{+, -\}$, depending whether v_i represents a positive example or a negative one.
Find $a_0, \ldots, a_n$ such that:

- the number of valuations v_i that satisfy $lab_i = +$ and $a_0 + a_1 * v_i(X_1) + \ldots + a_n * v_i(X_n) \leq 0$ is maximal, and
- the number of valuations v_i that satisfy $lab_i = -$ and $a_0 + a_1 * v_i(X_1) + \ldots + a_n * v_i(X_n) > 0$ is maximal.

In the following, $\mathcal{I}n_i^+$ denotes the inequality $a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) \leq 0$, whereas $\mathcal{I}n_i^-$ denotes the inequality $a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) > 0$. In these inequalities, $v_i(X_j)$ are numeric values whereas $a_0, \ldots, a_n$ become variables.

Example 4. Let us suppose that we have two variables: X_1 represents the radius of a circle and X_2 represents the length of a square. Let us consider the target concept, expressing the following relation between a circle and a square: "*the square can be drawn inside the circle*". The set of examples is defined by:
$\{[1,3,3,+], [2,3,3,+], [3,3,3,+], [4,3,3,+], [5,3,3,+], [6,3,3,+],$
$[7,3,3,-], [8,3,3,-], [9,3,3,-], [10,3,3,-], [11,3,3,-], [12,3,3,-]\}$
where a tuple $[i, c_1, c_2, l]$ represents the i-th example (v_i, lab_i) with $v_i = \{X_1/c_1, X_2/c_2\}$ and $lab_i = l$.

In the best case, we would like to find a_0, a_1, a_2 such that for each $i = 1..6$, $\mathcal{I}n_i^+$ is satisfied (for example $\mathcal{I}n_1^+ = a_0 + a_1 * 3 + a_2 * 3 \leq 0$) and for each $j = 7..12$, $\mathcal{I}n_j^-$ is satisfied (for example $\mathcal{I}n_7^- = a_0 + a_1 * 5 + a_2 * 2 > 0$).

A solution to $\mathcal{I}n_i^+ \cup \mathcal{I}n_i^-$ would give the equation of an hyperplane that separates the positive examples from the negative ones. Nevertheless, in many cases, this set has no solutions and we only aim at maximizing the number of inequalities that are satisfied. We solve this problem by using linear programming techniques, that can be expressed as follows:

$$\begin{cases} LP_g : \textit{maximize } \Sigma_{j=1}^n c_j x_j \\ \quad \textit{subject to} \\ \qquad \Sigma_{j=1}^n a_{ij} x_j \leq b_i, \ \textit{for } i = 1, 2, \ldots, m \\ \qquad l_j \leq x_j \leq u_j, \ \textit{for } j = 1, \ldots, n \\ \quad \textit{where } x_j \textit{ is a real or an integer.} \end{cases}$$

The problem $\mathcal{P}$ differs from the general LP_g problem for two reasons: first, our objective function is only to maximize the number of satisfied inequalities; secondly, strict inequalities appear in $\mathcal{I}n_j^-$.
Therefore, we replace the set $[\mathcal{I}n_i^+]_{\{i|lab_i=+\}}$ by the following set $[\mathcal{I}n_i''^+]_{\{i|lab_i=+\}}$:

$$\mathcal{I}n_i''^+ \colon a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) \leq \sigma_i * C,$$

where C denotes a fixed constant the value of which will be very high and where σ_i takes either the value 0 or 1. The underlying idea is that if the value of C is high enough, the inequality $a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) \leq \sigma_i * C$ will be satisfied with $\sigma_i = 1$.
In the same way we built the set $[\mathcal{I}n_i''^-]_{\{i|lab_i=-\}}$ containing the following inequalities:

$$\mathcal{I}n_i''^- \colon a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) \geq \epsilon - \sigma_i * C$$

where ϵ is a sufficiently low value which allows to remove the strict inequalities.
The set of inequalities $[\mathcal{I}n_i''^+]_{\{i|lab_i=+\}} \cup [\mathcal{I}n_i''^-]_{\{i|lab_i=-\}}$ contains $(n+1)+m$ variables, $a_0, \ldots, a_n, \sigma_1, \ldots, \sigma_m$. Let us recall that the number n is the number of numeric variables that have been introduced in the clause, whereas m is the number of positive and negative instances.

Minimizing $\Sigma_{i=1}^m \sigma_i$ enables to maximize the number of inequalities that are satisfied among $[\mathcal{I}n_i^+]_{\{i|lab_i=+\}} \cup [\mathcal{I}n_i'^-]_{\{i|lab_i=-\}}$.
The linear problem that is solved can thus be stated as follows:

$$\left\{\begin{array}{l} \mathit{minimize}\ \Sigma_{i=1}^m \sigma_i \\ \mathit{subject\ to} \\ \qquad a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) \leq \sigma_i * C,\ i = 1, \ldots, m\ \mathit{and}\ lab_i = +, \\ \qquad a_0 + \Sigma_{j=1}^n a_j * v_i(X_j) \geq \epsilon - \sigma_i * C,\ i = 1, \ldots, m\ \mathit{and}\ lab_i = - \\ \qquad a_j \in D,\ \mathit{for\ all}\ j = 0, \ldots, n, \\ \qquad \sigma_i \in \{0,1\},\ \mathit{for\ all}\ i = 1, \ldots, m. \end{array}\right.$$

5 Integration in ICC

As has already been mentioned, ICC is based on a top-down strategy: while negative examples are extensionally covered, the current clause is refined by adding a literal to its body. Let us suppose that the variables $X_1, \ldots, X_n$ have already been introduced in the clause. Each time a new numeric variable is introduced, the system ICC calls the system *lp_solve*, developed by M.R.C.M. Berkelaar[1]. When *lp_solve* succeeds in solving the associated linear problem, the values of σ_i enables to compute the number of positive instances and the number of negative instances that are $\mathcal{D}$-covered by the learned inequality which in turn, enable to compute the gain of this linear constraint. The literal that is introduced in the body of the clause is chosen, according to an entropy measure, among:

- the set of relevant constraints built by ICC,

[1] Eindhoven University of Technology, Eindhoven, The Netherlands

- the set of relevant constrained atoms built by ICC,
- if possible, the inequality $a_0 + a_1 * X_{j_1} + \ldots + a_p * X_{j_p} \leq 0$, which expresses a relation between the numeric variables $X_{j_1}, \ldots, X_{j_p}$ which have already been introduced in the clause.

In the system ICC, it is possible to introduce function symbols, as for instance the function *square* : $\mathbb{N} \rightarrow \mathbb{N}$. In this case, ICC searches for constraints c' : $a_0 + \Sigma_{i=1}^{m} a_i * X_i \leq 0$, where the variables X_i are either numeric variables which appear in the clause, or terms built with variables appearing in the clause.

Example 5. Let us consider, for instance, the concept $contains(O, R)$ meaning that *the object O can be drawn inside a circle of radius R* and let us consider the basic predicate $square(O, N)$ expressing that O is a square and that its length is N. If the function *sqr* (that computes the square of a number) is given, then ICC learns the following program:

$contains(a, b) \leftarrow square(a, b)$.
$contains(a, b) \leftarrow square(a, d)$, 0.92.sqr(b) -0.40.sqr(d) -5b + 1.91.d +5.82 > 0.

6 Conclusion

The system ICC is a system that builds, if possible, a constrained logic program, that $\mathcal{D}$-covers the positive examples and $\mathcal{D}$-covers no negative examples. It has been implemented in Sicstus Prolog on a Sun 4.

The experiments that we have made have already given positive results. Nevertheless, even if the system *lp_solve* that we use is efficient, so that searching for linear inequalities does not seem very expensive, the search space in ICC is, in general, very large due to the introduction of new terms. Biases have been introduced, as for instance the limitation of the depths of terms. Nevertheless, they are fixed before the learning process and we would like to study more dynamic biases linked to the learning problem and specifying terms or atoms that seem relevant.

Comparison with other works: Some systems already deal with the problem of learning constrained logic programs, as for instance, [11, 8]. In [11], the system combines a version space strategy with a divide-and-conquer strategy and the constraints that are learned mainly express bounds for variables. The strategy developed in [8] is an extension of Golem [7] to handle numeric constraints.

The problem of learning linear inequalities has already been a lot studied in the literature. It has been studied from a theoretical point of view, in [12]. There are two main differences with the Support Vector machines, defined in [12]: first, we intend to minimize the number of misclassified examples whereas their aim is to minimize the empirical risk; secondly, our method relies on Linear Programming techniques which is, as far as we know, original. This has also been studied in the field of learning decision trees, as for instance [1, 9]: generally, the underlying systems start from an initial inequation and perturb the coefficients until the impurity measure reaches a local minimum; in OC1 [9], a non deterministic step enables to get out the local minimum.

Finally, the idea of changing the feature space in order to learn non linear constraints has already been introduced in [12] and in [4]. Nevertheless, in our framework the functions that can be introduced in the equations depends on the underlying Constraint Logic Programming language, whereas in [4], the set of functions is reduced to the product between variables and to the square of a variable.

References

1. Breiman, Friedman, Olshem, Stone, 1993. Classification of Regression Trees. Chapman & Hall.
2. Bergère M., Ferrand G., Le Berre F., Malfon B., Tessier A., 1995. La programmation logique avec contraintes revisitée en termes d'arbres de preuve et de squelettes. Rapport de recherche 95-06, LIFO, université d'Orléans.
3. Chvátal V., 1983. Linear Programming. Freeman.
4. Ittner A., Schlosser M., 1996. Non-Linear Decision Trees - NDT. Proceedings of the 13th International Conference on Machine Learning (ICML 96), L. Saitta (Ed.), Bari, Italy.
5. Jaffar J., Maher M.J., 1994. Constraint Logic Programming: A Survey. Jal of Logic Programming, vol. 19/20, may/july 1994, pp. 503-581, Elsevier Science Publishing.
6. Martin L., Vrain Ch., 1996. Induction of contraint logic programs. Proceedings of the Algorithmic Learning Theory workshop, ALT 96, Sidney, Australia.
7. Muggleton S., Feng C., 1992. Efficient Induction of Logic Programs. Inductive Logic programming. The A.P.I.C. Series N^o 38, S. Muggleton (Ed.), Academic Press. pp. 281-298.
8. Mizoguchi F., Ohwada H., 1995. An Inductive Logic Programming Approach to Constraint Acquisition for Constraint-based Problem Solving. Proc. of the 5th Intl Workshop on Inductive Logic Programming, L. De Raedt (Ed.), pp. 297-323.
9. Murthy S., Kaisf S., Salzberg S., Beigel R., 1993. OC1: Randomized Induction of Oblique Decision Trees. Proceedings of the 11th National Conference on AI, AAAI-93, Cambridge, MIT Press, pp. 323-327
10. Quinlan J.R., 1990. Learning Logical Definitions from Relations. *Machine Learning Journal*, Vol. 5, Kluwer Academic Publishers, pp. 239-266.
11. Rouveirol C., Sebag M., 1995. Constraint Inductive Logic Programming. Proc. of the Fifth International Workshop on Inductive Logic Programming, L. De Raedt (Ed.), Leuven, September 1995, pp. 181-198.
12. Vapnik V.N., 1995. *The Nature of Statistical Learning Theory.* Springer.