Induction of constraint logic programs

Lionel MARTIN, Christel VRAIN
L.I.F.O. - Université d’Orléans - BP 6759
45067 Orléans Cedex 2, France
email: {martin,cv}@univ-orleans.fr

Abstract. Inductive Logic Programming is mainly concerned with the problem of learning concept definitions from positive and negative examples of these concepts and background knowledge. Because of complexity problems, the underlying first order language is often restricted to variables, predicates and constants. In this paper, we propose a new approach for learning logic programs containing function symbols other than constants. The underlying idea is to consider a domain that enables to interpret the function symbols and to compute the interest of a given value for discriminating positive and negative examples. This is modeled in the framework of Constraint Logic Programming and the algorithm that we propose enables to learn some constraint logic programs. This algorithm has been implemented in the system ICC. In order to reduce the complexity, biases have been introduced, as for instance the form of constraints that can be learned, the depth of a term or the size of the constraints.

1 Introduction

Inductive Logic Programming is mainly concerned with the problem of learning concept definitions, represented in first order languages, from positive and negative examples of these concepts and background knowledge. In most approaches, function symbols are either considered as pure syntactic entities or they are specified via the introduction of a new predicate symbol.

When considering function symbols as syntactic entities, the specification of the examples and/or of the background knowledge is complex; for instance, the positive examples of the predicate even have to be specified by:

(1) \{even(0), even(s(s(0))), even(s(s(s(s(0))))), \ldots \}

whereas it seems more natural to write:

(1') \{even(0), even(2), even(4), \ldots \}

with the underlying assumption that the domain of computation is the set of integers.

Moreover, such a specification has a great influence on the learned program: Specification 1 can lead to the learned clause \texttt{even(s(s(X)))} ↔ \texttt{even(X)} whereas the following specification of the positive examples

(2) \{even(+0, 0), even(+0, s(0)), even(+s(s(0)), s(0))), even(+s(s(0)), s(s(0)))), \ldots \}

would probably lead to the clause \texttt{even(+s(X), X)}. 
The other way to handle functions consists in replacing each n-ary function \( f \) by an associated \( n+1 \)-ary predicate \( p_f \) such that \( p_f(t_1, \ldots, t_{n+1}) \) is true iff \( t_{n+1} = f(t_1, \ldots, t_n) \). In this case, either the predicate \( p_f \) is extensionally specified and this specification can only be partial [10, 5], or it is intentionally specified and a partial semantics is computed [8]. In both cases, the incomplete definition of the function can prevent from learning correct programs. Furthermore, this approach leads to the problem of determinate literals for empirical learning systems which rely on an entropy measure, as the system Foil [10]; literals associated to functions do not allow to discriminate positive and negative examples and therefore, they are in general not introduced in the clause. In [3], the authors show that the program defining the predicate \( \text{quicksort}(t_1, t_2) \), cannot be learned, unless determinate literals are added to the body of the clause.

In this paper, we propose a new approach for learning logic programs containing function symbols in a Constraint Logic Programming framework. 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 order to reduce the potential large number of terms that can be interpreted by the same value, biases have been introduced in the implemented prototype ICC.

2 Basic definitions

2.1 Syntax

We briefly recall some basic notions about constraint logic programming, that can be found in [4] or in [2]. We consider:

- an infinite set of variables \( \mathcal{V} \),
- a set, denoted by \( \Sigma \), of function symbols,
- a set, denoted by \( \Pi_C \), of constraint predicate symbols,
- a set, denoted by \( \Pi_P \), of predicate symbols definable by a program.

Moreover, we suppose that \( \Pi_C \) contains the predicate symbol \( \rightarrow \), and the logic constants \( \text{True} \) and \( \text{False} \).

A term over \( \mathcal{V} \) and \( \Sigma \) is inductively defined as follows: a variable \( v \) of \( \mathcal{V} \) is a term and, if \( f \) is an n-ary function symbol of \( \Sigma \) 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 \( \Pi_C \) 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 \( \Pi_P \) 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 \(a \leftarrow b_1, \ldots, b_n\), where \(a\) is an atom and \(b_1, \ldots, b_n\) are either atoms or constraints; \(a\) is called the head of the constrained clause and the set \(b_1, \ldots, b_n\) is called the body of the constrained clause. In the following of this paper, we represent it by \(a \leftarrow c \sqcap b_1, \ldots, b_n\), 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 program that defines \(\text{factorial}\):
\[
\text{facto}(X, Y) \leftarrow X = 0, Y = s(0),
\]
\[
\text{facto}(X, Y) \leftarrow X = s(Z), Y = T \mathbin{\circ} X \mathbin{\circ} \text{facto}(Z, T).
\]
is a constrained program with \(\Sigma = \{0, s, \ast\}\), \(\Pi_C = \{=\}\) and \(\Pi_P = \{\text{facto}\}\).

Let \(C = A_0 \leftarrow c_1, \ldots, c_m \square A_1, \ldots, A_n\) be a constrained clause. The variable \(X\) is \(\mathcal{D}\)-linked in \(C\) if \(X\) occurs in the head of \(C\) or, if there is a primitive constraint \(c_i\) or an atom \(A_j\) that contains the variables \(X\) and \(Y\) and \(Y\) is linked in \(C\).
A constrained clause \(C\) is \(\mathcal{D}\)-linked if all its variables are \(\mathcal{D}\)-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 \((\Sigma, \Pi_C)\)-structure \(\mathcal{D}\) is composed of:

- a non-empty set \(D\),
- an assignment of a function \(f_D : D^n \rightarrow D\) for each \(f \in \Sigma\),
- an assignment of a function \(p_D : D^n \rightarrow \{\text{True}, \text{False}\}\) for each \(p \in \Pi_C\).

**Example 1:** (continued) Let us consider again the constraint program defining the function \(\text{factorial}\) with:
\[
\Sigma = \{0, s, \ast\}, \quad \Pi_C = \{=\}, \quad \Pi_P = \{\text{facto}\}.
\]
Let \(D = \mathbb{N}\) be the set of positive integers and let \(\mathcal{D}\) interpret the function 0 by the integer 0, the function \(s\) by the function \(\text{successor}\) on \(D\) and the function \(\ast\) by the multiplication. This defines a \((\Sigma, \Pi_C)\)-structure. \(\square\)

In the following, \(\mathcal{D}\) represents a \((\Sigma, \Pi_C)\)-structure on a domain \(D\).

Let \(t\) be a ground term. The interpretation of \(t\) 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 \(I_{\mathcal{D}}(f(t_1, \ldots, t_n)) = f_D(I_{\mathcal{D}}(t_1), \ldots, I_{\mathcal{D}}(t_n))\),

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 satisfies \(I_{\mathcal{D}}(t) = d\). This set is denoted by \(I^{-1}_{\mathcal{D}}(d)\).

An *inverse interpretation* of an expression \(p(d_1, \ldots, d_m)\), where \(p \in \Pi_C\) and \(d_i \in D\), w.r.t. \(T\), is the set of primitive constraints \(p(t_1, \ldots, t_m)\) with \(t_i \in I^{-1}_{\mathcal{D}}(d_i)\), \(i = 1, \ldots, m\).

For example, let \(D = \mathbb{N}\), \(\Sigma = \{0, 1, s, p\}\) and let \(T = \{0, s(0), s(s(0)), 1, p(s(1))\}\), \(I^{-1}_{\mathcal{D}}(1) = \{s(0), 1, p(s(1))\}\).
A $D$-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 $D$-atoms is called the $D$-base.

A valuation $v$ is a function: $V \rightarrow D$. 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 $v^{-1} = \{X_1/d_1, \ldots, d_n/X_n\}$ with $v(X_i) = d_i$.

Let $A = p(d_1, \ldots, d_n)$ be a $D$-atom and let $v$ be the valuation $\{X_1/d_1, \ldots, X_n/d_n\}$. If $v^{-1} \in V^{-1}(v)$, $v^{-1} = \{d_1/X_1, \ldots, d_n/X_n\}$, then $v^{-1}(A)$ denotes the atoms $p(X_1, \ldots, X_n)$. (The reader can easily check that the following relation holds: $v(v^{-1}(A)) = A$.)

For example, the inverse valuations of $v = \{X_1/a, X_2/b, X_3/a\}$ are $\{a/X_1, b/X_2, a/X_3\}$, $\{a/X_1, b/X_2, a/X_1\}$, $\{a/X_3, b/X_2, a/X_3\}$ and $\{a/X_3, b/X_2, a/X_1\}$.

3 Induction of constrained clauses

We describe in this section a general algorithm to build a constrained clause by iteratively adding either a constrained atom or a constraint in its body. The problem can be specified by:

1. a $(\Sigma, \Pi_C)$-structure $D$,
2. a set $\text{BASE}$ of basic predicates defined by a set, denoted by $\text{BASE}^+$, of $D$-atoms,
3. a set $\text{TARG}$ of predicates to learn, specified by two sets of $D$-atoms: $E^+$ and $E^-$ with $E^+ \cap E^- = \emptyset$. The set $E = E^+ \cup \neg E^-$ denotes the intended specification.

$\Pi_P$ denotes the set $\text{BASE} \cup \text{TARG}$.

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

The aim is to find a constrained program built over $\Sigma$, $\Pi_C$, and $\Pi_P$ which $D$-covers the positive examples and which $D$-covers no negative ones. In the field of Inductive Logic Programming, it has already been shown (see for instance [6]) that the notion of extensional coverage is not a sufficient condition for learning complete and consistent programs. This is still true when learning constrained logic programs. Nevertheless, in this paper we only propose a way of building constrained clauses based on the notion of $D$-coverage and we do not address the validation problem.

Example 2. Let us consider the following specification:
- $\Sigma = \{0, \text{pred}\}$, $\Pi_C = \{=\}$, and $D = \mathbb{N}$ (the set of positive integers),
- The $(\Sigma, \Pi_C)$-structure $D$ is the classical interpretation of 0 and $\text{pred}$ on the set of positive integers $\mathbb{N}$.
The basic predicate \textit{odd} is defined by \( 	extit{odd}^+ = \{\textit{odd}(1), \textit{odd}(3), \textit{odd}(5)\} \) and the target predicate \textit{even} is defined by \( E^+ = \{\textit{even}(0), \textit{even}(2), \textit{even}(4)\} \) and \( E^- = \{\textit{even}(1), \textit{even}(5)\} \) (the intended interpretation of \textit{even}(3) is unknown).

A possible solution is then:

\[
\begin{align*}
\textit{even}(X) & \leftarrow X = 0, \\
\textit{even}(X) & \leftarrow Y = \textit{pred}(<\textit{pred}(X)) \sqcap \textit{even}(Y).
\end{align*}
\]

We present in the remaining of the paper a method to build \( D \)-linked constrained clauses which \( D \)-cover some positive examples and no negative ones. The particularity of our approach is to handle function symbols as semantic entities instead of syntactic ones: in the previous example, \( \textit{pred} \) is interpreted as the function predecessor on the set of natural numbers. In Section 3.1, we present our algorithm, in Section 3.2, we explain the biases that have been implemented in the system ICC and Section 3.3 gives preliminary results.

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

### 3.1 Theoretical framework

The method used to build \( D \)-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 \( D \)-covered. We describe in this section the method that has been developed to build relevant constraints and relevant constrained atoms.

Since the aim of the construction is to build a clause which \( D \)-covers at least a \( D \)-uncovered positive example, the first step of the algorithm is to choose (randomly) a \( D \)-uncovered positive example \( e \in E^+ \) and to build a clause which \( D \)-covers \( e \) and as many \( D \)-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.

Let \( e \in E^+ \) be a \( D \)-uncovered positive example, and let \( C = A_0 \leftarrow c \sqcap A_1, \ldots, A_m \) be a clause under construction that \( C \) \( D \)-covers \( e \).

\textit{Computation of the possible constraints to add.} For each covering valuation \( v \) for \( e \), the constraint which can be added is any subset of the set of primitive constraints, denoted by \( \text{Const} \) and computed as follows:
1. Compute the set $T = \text{Term}_\Sigma(\text{Const}(v))$, 
2. Compute $D' = \{I_D(t) | t \in T\}$, 
3. Build $PC = \{p(d_1, \ldots, d_n) | p \in \Pi_C, d_i \in D', p_i(d_1, \ldots, d_n) \text{ is true in } D\}$, 
4. Compute $I_T^{-1}(PC) = \{I_T^{-1}(c) | c \in PC\}$, 
5. Remove trivial constraints from $I_T^{-1}(PC)$, 
6. Compute $\text{Constr} = \bigcup_{v^{-1}(v) \in \Sigma} \bigcup_{v^{-1}(v) \in \Sigma} I_T^{-1}(I_T^{-1}(PC))$

Algorithm 1.

In Example 2, the construction of the set $\text{Constr}$ for $e = \text{even}(2)$, $C = \text{even}(X) \leftarrow$ and $v = \{X/2\}$ can be summarized as follows:

\[
\begin{align*}
  v &= \{X/2\} \quad C = \text{even}(X) \leftarrow \quad \text{Constr} = \{0 = \text{pred}(\text{pred}(X))\} \\
  T &= \{0, 2, \text{pred}(0), \text{pred}(2), \ldots\} \\
  I_D &= \bigcup_{v^{-1}(v) \in \Sigma} \bigcup_{v^{-1}(v) \in \Sigma} I_T^{-1}(PC) = \{0 = \text{pred}(\text{pred}(2))\} \\
  D' &= \{0, 1, 2\} \quad \text{PC} = \{0 = 0, 1 = 1, 2 = 2\}
\end{align*}
\]

Let us just recall that $I_T^{-1}(PC)$ must contain only terms that belong to $T$. It explains why in this example, $I_T^{-1}(PC)$ does not contain, for instance, the constraint $1 = \text{pred}(2)$ ($1 \notin T$).

Computations of the possible constrained atoms to add: Let $X_1, \ldots, X_k$ be the variables occurring in $C$. For every $D$-atom $p(c_1, \ldots, c_n) \in (E^+ \setminus \{e\}) \cup BK^+$ and for every covering valuation $w$ for $e$, we build the set of constrained atoms $(c', p(X_{k+1}, \ldots, X_{k+n}))$, where $c'$ is computed by applying Algorithm 1 with:

- $v = w \cup \{X_{k+1}/c_1, \ldots, X_{k+n}/c_n\}$,
- $T = \text{Term}_\Sigma(\text{Const}(v)) \cup \text{Const}(p(c_1, \ldots, c_n))$,
- $I_T^{-1}(PC)$ is reduced to the primitive constraints where both a variable from $X_1, \ldots, X_k$ and a variable from $X_{k+1}, \ldots, X_{k+n}$ occur.

In Example 2, the set of possible constrained atoms built with $e = \text{even}(2)$ and $C = \text{even}(X_1) \leftarrow$ is:

\[
\begin{align*}
  &((X_2 = \text{pred}(\text{pred}(X_1))), \text{even}(X_2)) \quad \text{(obtained from even(0))}, \\
  &((X_2 = \text{pred}(X_1)), \text{odd}(X_2)) \quad \text{(obtained from odd(1))}, \\
  &((X_1 = \text{pred}(X_2)), \text{odd}(X_2)) \quad \text{(obtained from odd(3))}, \ldots
\end{align*}
\]

Once the set of all possible constraints and constrained atoms is built, their gain are computed and the constraint or the constrained atom with the best gain is added to the body of the clause.

### 3.2 Practical framework

The complexity of the previous algorithm is important for different reasons:
- the set $\text{Term}_\Sigma(\text{Const}(v))$ is generally infinite,
- if we allow some constraint predicates (such as $>$), the number of possible primitive constraints can become very large,
- in all cases, the number of possible primitive constraints when looking for constrained atoms can remain large.

In the system ICC, the form of constrained atoms is restricted as follows: Let $X_1,\ldots,X_k$ be the variables occurring in the clause under construction and let $p$ be a n-ary predicate in $\text{BASE} \cup \text{TARG}$. The primitive constraints that appear in the constrained atoms, $(c_i, p(X_{k+1},\ldots,X_{k+n}))$ have the form $X_{k+i} = t_i$ and $t_i$ is a term in which no variables from $X_{k+1},\ldots,X_{k+n}$ occurs.

Furthermore, different biases have been introduced:

- limitation of the depth of the terms,
- limitation of the number of primitive constraints in constraints,
- association of a mask to function symbols: if $g$ belongs to the mask of a function symbol $f$, then a term $f(t_1,\ldots,t_n)$ with $t_i = g(\ldots)$ is not considered. This bias avoid to build terms such as $\text{pred}(\text{suc}(t))$.
- possible introduction of "user-defined" function symbols specified by their definition in a functional language: this allows to define functions such as $\text{partition}\_\text{sup}$ used for learning quick-sort.

3.3 Examples

We give some programs learned by ICC, respectively defining $\text{factorial}$, $\text{member}$ and $\text{quick-sort}$:

| Learned program                                      | $|L^+|\ |L^-|$ |
|------------------------------------------------------|-----------|
| $\text{factorial}(X,Y) \leftarrow X \equiv 0, Y = \text{suc}(0)$, | 7         |
| $\text{factorial}(X,Y) \leftarrow Z = \text{pred}(X), T = \text{div}(Y, X) \triangleq \text{factorial}(Z, T)$. | 12        |
| $\text{member}(X,Y) \leftarrow X \equiv \text{head}(Y)$. | 7         |
| $\text{member}(X,Y) \leftarrow Z \equiv X, T = \text{body}(Y) \triangleq \text{member}(Z, T)$. | 16        |
| $\text{qs}(X_1, X_2) \leftarrow X_1 = [], X_2 = []$. | 6         |
| $\text{qs}(X_1, X_2) \leftarrow (X_3 = \text{part}\_\text{sup}(\text{head}(X_1)), \text{body}(X_1)), \text{qs}(X_3, X_4)$, | 16        |
| $(X_5 = \text{part}\_\text{inf}(\text{head}(X_1), \text{body}(X_1)), \text{qs}(X_5, X_6)$, | 20        |
| $(X_2 = \text{append}(X_6, \text{cons}(\text{head}(X_1), X_4)))$. | 16        |

Let us notice that quick-sort has been learned with the user-defined functions $\text{part}\_\text{inf}$, $\text{part}\_\text{sup}$ and $\text{append}$.

4 Conclusion

We propose in this paper a method for learning constrained clauses: this method has been developed to learn logic programs containing function symbols which are considered as semantic entities instead of syntactic ones. Moreover, this gives an interesting solution to the problem of determinate literals.

The particularity of the approach consists in considering non-Herbrand interpretations for the function symbols; then the definitions given in the previous
examples cannot be learned by most ILP techniques such as [8, 7, 11]. However, the complexity of the constraints, and particularly the complexity of numerical constraints, is limited by the depth of the terms. Practically, the depth can not be greater than 3 or 4 but, on the other hand, non-linear numerical constraints can be built even if their form remain simple. For the time being, ICC is not able to learn constraints where numerical values appear such as $X \leq 18$, which is possible in other works [7, 11]. An extension of the system ICC to learn such simple numerical constraints is under progress.

References


This article was processed using the BiBTeX macro package with LLNCS style