Applying MML to ILP[*]

D. L. Dowe
CSSE, Clayton School of I.T.
Monash University
3168 Clayton, Australia
David.Dowe@infotech.monash.edu.au

Cèsar Ferri[†]
Dept. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
46022 Valencia, Spain
cferri@dsic.upv.es

José Hernández-Orallo
Dept. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
46022 Valencia, Spain
jorallo@dsic.upv.es

María José Ramírez Quintana
Dept. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
46022 Valencia, Spain
mramirez@dsic.upv.es

Abstract

In Inductive Logic Programming (ILP), since logic is a complete (universal) language, infinitely many possible hypotheses are compatible (hence plausible) given the evidence. An intrinsic way of selecting the most convenient hypothesis from the set of possible theories is not only useful for model selection but it is also useful for guiding the search in the hypotheses space, as some ILP systems have done in the past. One selection/search criterion is to apply Occam's razor, i.e. to first select/try the simplest hypotheses which cover the evidence. In order to do this, it is necessary to measure how simple a theory is. The Minimum Message Length (MML) principle is based on information theory and it reflects Occam's razor philosophy. In this paper we present a MML method for costing both logic programs and sets of facts according to the theory. Our scheme has a solid foundation and avoids the drawbacks of previous coding schemes in ILP, namely the model complexity and proof complexity approaches.

1 Introduction

Inductive Logic Programming (ILP) [7] is currently a very important area of research as an appropriate framework for the inductive inference of first-order clausal theories from facts. ILP has provided an outstanding advantage in the inductive machine learning field by increasing the applicability of learning systems to theories with more expressive power than propositional frameworks.

Given a set of facts, ILP methods usually infer a set of different hypotheses that cover (partially or totally) the positive facts and refute (totally or partially) the negative examples. Therefore, a good option to select hypotheses, as well as an evaluation and a search method, is to follow Occam's Razor and prefer simple hypotheses. For this purpose we need to measure the complexity of the learnt program with respect to the evidence.

Several coding methods for ILP have been previously presented [1],[6]. In these approaches, the evidence is composed of positive

[*]This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN 2004-7943-C04-02, Generalitat Valenciana under grant GV06/301 and UPV under grant TAMAT.

[†]C. Ferri was supported by grant 2765 of UPV during a stay at Monash University.

examples only[1]. However there are important drawbacks in these schemes: the coding in [6] can be counter-intuitive for programs that perfectly cover the evidence and the coding in [1] cannot be applied if the program signature contains function symbols with arity greater or equal to 1. We discuss these problems in the following sections.

In this paper, we present an alternative coding scheme based on the Minimum Message Length (MML) principle. MML [13] is a formal information theory restatement of Occam's Razor: even when models are not equal in explaining the observed data, the model generating the shortest overall message (data and model) is more likely to be correct. Our scheme is based on applying some of the MML techniques for defining a coding method for logic programs, and then a coding method of the evidence with respect to the model.

The paper is structured as follows. Section 2 includes a brief description of the MML principle. Section 3 reviews and discusses two previous approaches for coding the evidence, and then presents our evidence coding scheme. Section 4 presents the coding for logic programs, and it includes a simple example. Section 5 contains two examples of how our scheme can be applied. The paper finishes with some conclusions and future work.

2 Minimum Message Length

The Minimum Message Length (MML) principle of inductive inference [16, 13, 15], is based on information theory, and hence lies on the interface of computer science and statistics. A Bayesian interpretation of the MML principle is that it variously states that the best conclusion to draw from data is the theory with the highest posterior probability or, equivalently, that theory which maximises the product of the prior probability of the theory with the probability of the data occurring in light of that theory. this immediately below.

Letting E be the data and T be a theory with prior probability $\Pr(T)$, we can write the posterior probability $\Pr(T|E) = \Pr(T \wedge E)/\Pr(E) = \Pr(T) \cdot \Pr(E|T)/\Pr(E)$, by repeated application of Bayes's Theorem. Since E and $\Pr(E)$ are given and we wish to infer T, we can regard the problem of maximising the posterior probability, $\Pr(T|E)$, as one of choosing T so as to maximise $\Pr(T) \cdot \Pr(E|T)$.

An information-theoretic interpretation of MML [13] is that elementary coding theory tells us that an event of probability p can be coded (e.g. by a Huffman code) by a message of length $l = -\log_2 p$ bits.

So, since $-\log_2(\Pr(T) \cdot \Pr(E|T)) = -\log_2(\Pr(T)) - \log_2(\Pr(E|T))$, maximising the posterior probability, $\Pr(T|E)$, is equivalent to minimising

$$MessLen = -\log_2(\Pr(T)) - \log_2(\Pr(E|T))$$

the length of a two-part message conveying the theory, T, and the data, E, in light of the theory, T. Hence the name "minimum message length" (principle) for thus choosing a theory, T, to fit observed data, E. For a comparison with the related subsequent Minimum Description Length (MDL) work of Rissanen [10, 11], see, e.g., [15] and other papers in that special issue of the *Computer Journal* and Chapter 10 of [13].

3 Evidence Representation and Coding

Given the MML philosophy it seems straightforward how to find the best theory T for an evidence E (and a prior). We would select the one such that:

$$Cost_{MML} = L(T) + L(E|T) \qquad (1)$$

is minimised. So, we only need to determine a way to cost theories such as T and to cost the evidence E wrt. T. However, there are many ways to do this and most of them are inefficient. Hence, it is not so obvious how to apply the MML philosophy to logic programs. In this section, we will analyse how $L(E|T)$ can be measured and how it depends on the way the evidence is represented, more specifically, the presence of repeated examples in the evidence.

[1]Logic programs are learnable from positive examples only as [5] shows.

3.1 The classical approaches to measure $L(E|T)$

Basically, the term $L(E|T)$ measures how E differs from what can be inferred from the theory T. In order to give a more precise definition of $L(E|T)$, first we have to define formally the data "that can be inferred from the theory" or, in other words, the empirical content of the theory. Following [1], we define the empirical content of a theory T, $Q(T)$, as:

$$Q(T) = O \cap M(T)\ ,\ \ E \subseteq O \tag{2}$$

where O are the possible observations (this excludes auxiliary predicates and concentrates on observable predicates and constants), and $M(T)$ is the least Herbrand model of the logic program T (i.e., the set of ground logical consequences of T). The measure of [1] is based on the size of $Q(T)$ (that is, the size of a subset of the least Herbrand model). Hence, this approach is called Model Complexity ($\mathcal{MC}$). A similar approach can be found in [4].

Using the above notation, three possible situations are distinguished in [1]:

1. $E = Q(T)$. The theory covers all and only all the examples.
2. $E \not\subseteq Q(T)$. There are examples not covered by the theory.
3. $E \subseteq Q(T)$. The theory covers all the examples, and perhaps other observable atoms.

Obviously, the first case is ideal but not very frequent. Case 2 is not considered in [1] because they argue that the theory can always be augmented in the following way:
$T' = T + (E - Q(T))$

In the rest of the section, we assume that T is the augmented theory T'. Now, it is only necessary to assume that $M(T)$ is finite (if no function symbols are allowed this is always the case with finite constant symbols) and then they employ this simple method:

$$L_{\mathcal{MC}}(E|T) = \log_2 \binom{|Q(T)|}{|E|} \tag{3}$$

Where $(\cdot)$ means combinatorial. There are, however, some problems with this approach:

1. It cannot be applied when $Q(T)$ is infinite. This happens frequently if we have function symbols.
2. The larger the evidence, the larger the term $L_{\mathcal{MC}}(E|T)$ grows (unless $Q(T) = E$), even if $Q(T)$ and E are very similar.
3. As expected, adding a new positive example to the evidence would need fewer bits than adding an example as a rule to the theory, but this asymmetry depends highly on the size of the evidence.

Apart from the $\mathcal{MC}$ approach, there is a different approach, called the "proof complexity" measure ($\mathcal{PC}$) [6], defined as the bits required to code the proof of each example given the theory. Here,

$$L_{\mathcal{PC}}(E|T) = \sum_{A \in E} L_{\mathcal{PC}}(A|T) \tag{4}$$

In this case, only the given evidence is coded, never the absent examples (the exceptions). This is counter-intuitive, since $L_{\mathcal{PC}}(E|T) > 0$ when $Q(T) = E$. Even with a perfect-covering program, this is not zero.

3.2 A new approach to measure $L(E|T)$

In order to take the best from both approaches ($\mathcal{MC}$ and $\mathcal{PC}$), we change our point of view to a more classical probabilistic approach. We just try to derive the probability of each possible evidence and then obtain the cost of coding it as the $-\log_2$ of the probability.

We define the Evidence Complexity approach ($\mathcal{EC}$) as:

$$L_{\mathcal{EC}}(E|T) = -\log_2 p(E|T) \tag{5}$$

where $p(E|T)$ represents the probability of seeing the evidence given the program T. The previous formula does not solve the problem by itself, since there can be many different ways to estimate $p(E|T)$. This approach is closely realted to Stochastic Inductive Logic Programming [4, 2, 8].

The idea is to use the program as a stochastic example generator. This is highly related to the $\mathcal{PC}$ approach, but we have to derive the probabilities with some conditions.

For instance, let us consider the following program P_1[2]: $\mathtt{r}_1 : \mathtt{even(0)}.$ $\mathtt{r}_2 : \mathtt{even(s(s(X))) :- even(X)}.$ Here, we could assign a uniform probability to each rule with the same predicate on the head. So, r_1 could have probability 0.5 and r_2 could have probability 0.5. Now, from here we can derive the probability of each possible observable fact or consequence of the program: $p(\mathtt{even(0)}) = 0.5$, $p(\mathtt{even(s(s(0)))}) = 0.25$, $p(\mathtt{even(s(s(s(s(0)))))}) = 0.125$, ... Here, it is the case that $\forall e \in Q(T) : p(e|T) > 0$ and also that

$$\sum_{e \in Q(T)} p(e|T) = 1 \qquad (6)$$

In case this second condition does not hold we can normalise the probabilities.

When coding programs with respect evidence, we should consider that the evidence could have or not repeated examples. In addition, we can consider programs that cover only the evidence, or programs that cover other facts that are not at the evidence. As we have commented, we do not consider cases where some examples are not covered by the program, in that situations we only need to add the not-covered examples to the program. Therefore we have four different settings:

- **No repeated examples and** $E = Q(T)$: This is the simplest case, since $Q(T)$ cover all example in E and nothing else, from the set of logic consequences of $Q(T)$ we will obtain E, and therefore we do not need to transmit $L(E|T)$.
- **No repeated examples and** $E \subset Q(T)$: In this case we have two options: To code E with respect T, or to code the exceptions, i.e. $Q(T) \notin E$. We select the cheapest option and then we add an aditional bit to inform about our selection.
- **Repeated examples and** $E = Q(T)$: From $Q(T)$ we can deduce E, but in this situation some of the elements of E appears more than once, therefore we need to code the number that every element appears (N_e), this costs $\sum_{e \in E} log^*(N_e)$.
- **Repeated examples and** $E \subset Q(T)$: This case is very similar to the previous one, but here we have that some logical consequences of T does not appear in E, then we need to code E wrt. T. Another option could be to code the number that every element appears (N_e) where N_e can be 0, this costs $\sum_{e \in E} log^*(N_e + 1)$.

Given these ways of coding the evidence wrt. the theory, we can then tackle the issue of assigning probabilities to evidences, i.e., to sets of examples. In our proposal, we just consider that the evidence doesn't contain repeated examples. [3] introduces a different costing method based on the premise that the evidence contains repeated examples.

3.2.1 Coding with no repeated examples

Let us consider a set of examples E and a program T. Then,

- First, code the length of the evidence[3]: $\log^*(|E|)$.
- Second, we compute the probability of the evidence ($\log_2 p_{norep}(E|T)$) as the product of the single probabilities, but we have to renormalise the probabilities each time a new example is extracted.

Given an evidence $E = \langle e_1, e_2, e_3, \ldots, e_n \rangle$, we denote by $\Pi = \{\langle e'_1, e'_2, e'_3, \ldots, e'_n \rangle\}$ the set of all possible permutations of elements from E without repetition. Then, $p_{norep}(E|T) = p_{norep}(E|\mathcal{P}) = \sum_{p \in \Pi} p(e'_1) \cdot p_{norep}(p_{2..n}|\mathcal{P}')$, where $p_{i..j} = \langle i, i+1, \ldots, j \rangle$, $i \leq j$. Also, $\mathcal{P}$ denotes the probabilities inferred from T, whereas $\mathcal{P}'$ are the corrected probabilities in the following way. If e'_i is the example extracted, then

[2]The 0 and s function symbols represents the constant zero and the successor function (respectively) over natural numbers.

[3]$\log^*(n)$ represents the bits needed to code the integer n in an efficient way. See [13] for details.

$$p(e'_{i+1}) = \frac{p(e'_{i+1})}{1-p(e'_i)} \cdots p(e'_n) = \frac{p(e'_n)}{1-p(e'_i)}$$

Overall, we have that

$$L_{\mathcal{EC}-norep}(E|T) = \log^*(|E|) - \log_2 p_{norep}(E|T) - \log_2(\prod_{p \in P_E} card(p)!)$$

where P_E is the set of different probabilities of examples from E and $card(p)$ is the number of examples from E whose probability is p. Let us show an example. Suppose that there are 5 probabilities (p_1, p_2, p_3, p_4 and p_5) of 5 events (e_1, e_2, e_3, e_4 and e_5), and $\sum_{i=1}^{5} p_i = 1$. Suppose that we wish to choose 3 of these: e.g. e_2, e_3 and e_5. We can send a code of length $\log^*(3+1) = log^*(4)$ to specify that we are choosing 3 events. Including order, there are $P_4^5 = 5 \times 4 \times 3 = 60$ different ways to choose 3 (ordered) objects. The sum of the probabilities of the 60 different ordered ways must add up to 1. Ignoring order, there are $C_3^5 = \frac{(P_3^5)}{3!} = \frac{(5\times4)}{2} = 10$ different ways to choose 3 (unordered) objects.

Then, the probability of the 3 events e_2, e_3 and e_5 is the sum of the 6 possible orders:

$$p_{norep}(E|T) =$$

$$\frac{p_2 \times p_3}{(1-p_2)} \times \frac{p_5}{(1-p_2-p_3)} + \frac{p_2 \times p_5}{(1-p_2)} \times \frac{p_3}{(1-p_2-p_5)} +$$
$$\frac{p_3 \times p_2}{(1-p_3)} \times \frac{p_5}{(1-p_3-p_2)} + \frac{p_3 \times p_5}{(1-p_3)} \times \frac{p_2}{(1-p_3-p_5)} +$$
$$\frac{p_5 \times p_2}{(1-p_5)} \times \frac{p_3}{(1-p_5-p_2)} + \frac{p_5 \times p_3}{(1-p_5)} \times \frac{p_2}{(1-p_5-p_3)}$$

If all the events have the same probabilities, then the probability is very easy to compute. In this case $p_{norep}(E|T) = \frac{1}{C_3^5} = \frac{3!}{(P_3^5)} = \frac{1}{10}$.

4 Costing Logic Programs

In this section we present our coding scheme for logic programs. The presented coding is similar to [12]. We consider four steps in our scheme: first, we encode information about predicates and functions, then rule heads, rules bodies and, finally, we encode the links among repeated variables. Therefore, given a program P with m rules $r_1, r_2, \ldots, r_m$, the cost of encoding P is:

$$cost(P) = \log^*(m) + cost(\Sigma_P) + \sum_{1 \le i \le m} cost(r_i) \tag{7}$$

The first factor assumes that the program is not empty. $cost(\Sigma_P)$ represents the cost of the information about predicates and functions contained in the signature Σ_P of the program. For each rule r of the form $H : -B.$, its cost is calculated as $cost(r) = cost(H) + cost(B) + cost(V_r)$, where the last term represents the cost of coding the variables V_r appearing in the rule.

4.1 Costing information of functions and predicates, $cost(\Sigma_P)$

Given a program P with n_p predicate and n_f function symbols, we need $\log^*(n_p) + \log^*(1+n_f)$ bits for encoding this information. Note that we add 1 to the number of function symbols because we can find non-empty programs without function symbols. Next, for each predicate symbol p_i we need $\log^*(arity(p_i))$ bits for coding its arity. For the functions, it is very similar, but in this case we can find functions with arity 0, then for each function symbol f_i we need $\log^*(arity(f_i)+1)$ bits.

Now, the names of the function and predicate symbols were arbitrary in that changing the order of the names would not change anything, so we can correspondingly subtract from the message length. Another way to think of this is in terms of the equivalence of the paradigms of probability and message length via $p_i = 2^{-l_i}$ and $l_i = -\log_2 p_i$, as emphasised in [14]. If we have several syntactically different ways of encoding something semantically equivalent, then the probability of the event increases as a result of the summation and the message length correspondingly decreases. The corresponding subtraction to make from our message length is: $-\log(n_f!) - \log(n_p!)$

Finally, the costing information of functions and predicates is calculated as:

$$cost(\Sigma_P) = \log^*(n_p) + \log^*(1+n_f) + \sum_{1\le i\le n_p} \log^*(arity(p_i)) - \log(n_p!) + \sum_{1\le i\le n_f} \log^*(1+arity(f_i)) - \log(n_f!) \tag{8}$$

4.2 Costing rule heads, $cost(H)$

For coding the heads of the rules of a logic program, first we need to tell which predicate symbol appears in each head. Here, we consider each of the m rules costing $\log_2(n_p)$, totalling $m \times \log_2(n_p)$.

We now come to the issue of encoding rule heads, given that just above we have encoded the relevant predicate symbols. There are issues of encoding variables and function symbols in the rule heads. Our simple method is to have a probability of 1/2 each for both variables and function symbols. That is, we need $\log_2(2)$ bits to set whether it is a variable or a function symbol.

When we have a function in the head, we must add $\log_2(n_f)$ since we need to code which function it is (we consider uniform distribution of probabilities). If its arity is greater than 0, then we can continue recursively.

4.3 Costing rule bodies, $cost(B)$

In the next step we encode the bodies of the rules. Due to execution considerations of logic programs, it is assumed both that the order of the rules matters and that the order of the literals in the bodies of the rules matters.

For each rule, encode the number of literals n_l with $\log^*(1+n_l)$. For each literal we need $\log_2(n_p)$ bits to determine their predicate symbol. Now, for each argument we use $\log_2(2)$ to indicate whether it is a variable or a function. If it is a function, we need $\log_2(n_f)$ to inform which function it is. If the function has arity greater than 0 we continue in the same way with its arguments.

4.4 Costing rule variables, $cost(V_r)$

With regard to variables, the receiver, after the previous information is sent, can know the number of variables existing in the rule, however, we also need to know if there are repeated variables and their location in the rule.

If there were d positions and n_v variables and no requirement that any variable had to be mentioned more than once, then there would be $(n_v)^d$ possibilities with a probability of $1/((n_v)^d)$ each and a code length for each one of $-\log(1/(n_v^d)) = d \times \log_2(n_v)$. However, we know that $1 \le n_v \le d$ and that each of the n_v variables has to appear in at least one position.

As an example, suppose $d = 10$ and $n_v = 3$. Therefore we have $n_v^d = 3^{10} = 59049$ possible combinations, but from these we have to remove the combinations that does not contain the 3 variables. Let us explore which these useless combinations are. The number of ways that one variable of the three can fill all 10 positions is 3. The number of ways that two variables of the three variables can fill all 10 positions is $\binom{3}{2} \times (2^{10} - 2) = 3 \times 1022 = 3666$. Therefore, the number of ways that three variables can fill all 10 positions with each of the three appearing at least once is 3^{10} minus the sum of the ways it can be done with the pairs using at least one of each minus the sum of the ways it can be done with exactly one $= 3^{10} - \binom{3}{2} \times (2^{10} - 2) - 3 \times 1 = 55980$. Generalising,

$$F(d, n_v) = (n_v)^d - \sum_{1\le i\le(n_v-1)} \binom{n_v}{i} * F(d,i) \tag{9}$$

where $F(d,0) = 1$. Note that, $F(d,d) = d!$ and if $d >> n_v$, $F(d,n_v) \approx (n_v)^d$.

Summing up, since the receiver already knows for each rule the number of variable positions d, we only need to transmit n_v, but $1 \le n_v \le d$, then :

$$cost(V_r) = \log_2(d) + \log_2(F(d,n_v))$$

4.5 A logic program coded with our scheme

Consider the following deterministic logic program P:

1	$even(0).$
2	$even(X) : -pos(X), even(Y), sum(X, s(s(0)), Y).$
3	$pos(s(X)).$
4	$sum(X, 0, X).$
5	$sum(X, s(Y), s(Z)) : -sum(X, Y, Z).$

For encoding this program with our scheme, we begin encoding the number of rules: $\log^*(5)$. Next, we encode the information of function and predicate symbols, $cost(\Sigma_P)$, as we have stated in Section 4.1. Now, the number of function and predicates symbols in P is $n_f = 2$ and $n_p = 3$, respectively. Also, we have to consider the arities of the symbols. For predicate symbols, *even* and *pos* have arity 1, and *sum* has arity 3. For function symbols, 0 has arity 0 and s has arity 1. Then, applying Formula 8 we obtain

$$\begin{aligned}
&cost(\Sigma_P) = \\
&\log^*(3) + \log^*(1+2) + \textstyle\sum_{1 \le i \le 3} \log^*(arity(p_i)) + \\
&\textstyle\sum_{1 \le i \le 2} \log^*(1 + arity(f_i)) - \log(3!) - \log(2!) \\
&= \log^*(3) + \log^*(3) + \log^*(1) + \log^*(1) + \log^*(3) \\
&+ \log^*(1+0) + \log^*(1+1) - \log_2(6) - \log_2(2)
\end{aligned}$$

4.5.1 Heads of Rules

The next step it to encode the heads of all the rules. First, we must encode the predicate symbols in the rule heads. In the example, the predicate symbols appear in the rule heads in the order *even*, *even*, *pos*, *sum*, *sum*. This costs $5 \times \log_2(3) = \log_2(3^5) = \log_2(243)$ to encode.

The cost of encoding the heads of the rules is shown in the following Table. Here, we can see how many bits it costs to encode each one of the parameters, as well as the cost of the variables.

Rule	Param1	Param2	Param3	Total
1	$2\log_2(2)$			$\log_2(4)$
2	$\log_2(2)$			$\log_2(2)$
3	$3\log_2(2)$			$\log_2(8)$
4	$\log_2(2)$	$2\log_2(2)$	$\log_2(2)$	$\log_2(16)$
5	$\log_2(2)$	$3\log_2(2)$	$3\log_2(2)$	$\log_2(128)$

4.5.2 Body of Rules

The following table illustrate the cost of encoding the bodies of the rules:

Rule	n_1	Literals	Total
1	$\log^*(1)$		$\log^*(1)$
2	$\log^*(4)$	$3\log_2(3)+$ $10\log_2(2)$	$\log_2(27648) + \log^*(4)$
3	$\log^*(1)$		$\log^*(1)$
4	$\log^*(1)$		$\log^*(1)$
5	$\log^*(2)$	$\log_2(3)+$ $3\log_2(2)$	$\log_2(24) + \log^*(2)$

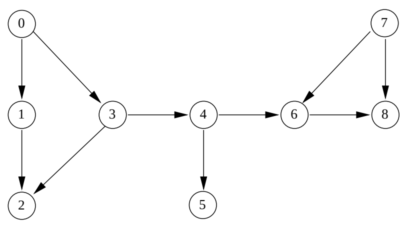

Figure 1: Network representing the reachability of the nodes.

4.5.3 Variables:

The first rule has no variables, so we start with the second one. First, we need to transmit the number of distinct variables n_v, since we have 5 options (1, 2, 3, 4 or 5 different variables) then $\log_2(5)$ and the information about their distribution in the 5 possible places, $\log_2(F(5,2)) = \log_2(30)$. The following table includes the bits required for encoding this information for all the rules:

Rule	2	4	5
Bits	$\log_2(150)$	$\log_2(2)$	$\log_2(3240)$

5 Coding Examples

In this section, we show how can we apply our approach to select the best hypothesis when we have some available.

5.1 Network

The first example was also employed in [1], although it is original from [9]. The goal of this problem is to learn the predicate "reach", that expresses the binary "reachability" relation in a directed graph. One vertex can reach another if there is a path between them in the graph. The network of this problem is shown in Figure 1.

The signature comprises two binary predicates *reach* and *linked*, along with nine constants {0,..., 8}. The background knowledge contains an extensional definition of the predicate *linked* for the network of the Figure 1: $\{<0,1>.\ <0,3>.\ <1,2>.\ <3,2>.\ <3,4>.\ <4,5>.\ <4,6>.\ <6,8>.\ <7,6>.\ <7,8>.\}$

The notation $< x, y >$ is shorthand for expressing the fact $linked(x, y)$. This fact means that there is a directed edge between vertexes x and y in the network. Below we ignore the complexity of this background knowledge, as it is constant for all proposed theory. The complete example set E for this predicate *reach* is:

$$\{(0,1).(0,2).(0,3).(0,4).(0,5).(0,6).(0,8).(1,2).(3,2).$$

$$(3,4).(3,5).(3,6).(3,8).(4,5).(4,6).(4,8).(6,8).(7,6).(7,8).\}$$

The notation (x, y) is shorthand for expressing the fact $reach(x, y)$. Given this evidence and the background knowledge, consider that we have six different theories that cover E. The following table includes these theories:

1	$reach(X,Y).$
2	$reach(0,1).$ $reach(0,2).$ $reach(0,3).$ $reach(0,4).$ $reach(0,5).$ $reach(0,6).$ $reach(0,8).$ $reach(1,2).$ $reach(3,2).$ $reach(3,4).$ $reach(3,5).$ $reach(3,6).$ $reach(3,8).$ $reach(4,5).$ $reach(4,6).$ $reach(4,8).$ $reach(6,8).$ $reach(7,6).$ $reach(7,8).$
3	$reach(X,Y)\ :-linked(X,Y).$ $reach(0,2).$ $reach(0,4).$ $reach(0,5).$ $reach(0,6).$ $reach(0,8).$ $reach(3,5).$ $reach(3,6).$ $reach(3,8).$ $reach(4,8).$
4	$reach(X,Y)\ :-linked(X,Y).$ $reach(X,Y)\ :-linked(X,Z).$
5	$reach(X,Y)\ :-linked(X,Y).$ $reach(X,Y)\ :-linked(X,Z),\ \ linked(Z,Y).$ $reach(0,5).$ $reach(0,6).$ $reach(0,8).$ $reach(3,8).$
6	$reach(X,Y)\ :-linked(X,Y).$ $reach(X,Y)\ :-linked(X,Z),\ \ reach(Z,Y).$

Theory T_1 is the most general theory T, only one general rule. T_2 is the least general theory: the 19 examples expressed as facts. The first rule of theory T_3 states that two vertexes are reachable if they are linked; since this rule alone is incomplete, we add the not covered facts as rules (see Section 3.1). Theory T_4 has a too general rule which states that vertexes X and Y are reachable if X is linked to some vertex. Theory T_4 has a redundant rule. The first two rules of theory T_5 cover all but 4 examples: again, we add these 4 examples as rules. Finally, theory T_6 expresses the reachability relation using a recursive rule.

Table 1 presents the code lengths of these programs using our approach. The first 5 columns contains the cost in bits of coding some parts of the theories. The sum of these parts (seventh column) represents the cost of coding only T (see 4). The eighth column includes the cost of expressing E with respect T for the programs that cover more than E, i.e. $L(E|T)$. We have also included the option of coding the exceptions instead of E (ninth column). We can select the cheapest adding 1 bit in order to inform about our choice. Finally, column 9 $(L(T) + L(E|T))$ contains the total cost in bits of every program.

As expected, if we observe the cost of the theories, the most general theory T_1 and the most specific theory T_2 are the least and most complex theories, respectively. But when we consider also the cost of coding E with respect to T, the order is (from cheapest to most expensive): $T_6 > T_1 > T_5 > T_4 > T_3 > T_2$. The following table compares the evaluation of each theory according to the Model Complexity ($\mathcal{MC}$) approach, the Proof Complexity $L_{\mathcal{PC}}$ approach[4], and our approach the Evidence Complexity approach ($\mathcal{EC}$): The ranking of theories given by each measure is as follows:

$\mathcal{MC}$	T_6	T_1	T_4	T_5	T_3	T_2
$\mathcal{PC}$	T_1	T_4	T_6	T_5	T_3	T_2
$\mathcal{EC}$	T_6	T_1	T_5	T_4	T_3	T_2

While $\mathcal{MC}$ and $\mathcal{EC}$ obtain almost identical rankings, $L_{\mathcal{PC}}$ differs in their preferences. Both ($\mathcal{MC}$ and $\mathcal{EC}$) rank the "correct" theory (T_6) as the best, and T_2 as the worst. $L_{\mathcal{PC}}$ selects the most general theory T_1 as the best theory. This fact is mainly because $L_{\mathcal{PC}}$ needs to encode always E, even when it can be derived from T. For that reason, it gives preference to short theories even though being too general.

5.2 Sum

For the second example we are dealing with the problem of summing natural numbers. On the contrary of the last example, O (possible observations) is infinite since there are infinite natural numbers. In this situation, the $\mathcal{MC}$ approach cannot applied because it can not

[4]We employ the results from [1].

T	#Rul.	Lexic.	Heads	Bod.	Vbles	L(T)	L(E|T)	L(E|T)	L(T)+
							E	Except.	L(E|T)
1	1.52	9.12	3.00	1.52	1.00	16.16	70.39	73.25	86.55
2	9.00	9.12	177.46	28.85	0.00	224.43	0.00	0.00	224.43
3	7.36	9.12	91.06	34.37	5.81	147.72	0.00	0.00	147.72
4	2.52	9.12	10.00	26.22	12.48	60.34	70.39	87.98	130.73
5	5.93	9.12	47.36	34.55	17.47	114.42	0.00	0.00	114.42
6	2.52	9.12	10.00	28.47	17.47	67.58	0.00	0.00	67.58

Table 1: Cost in bits of every theory.

deal with infinite observables. The evidence is composed by the following set of examples: $sum(s(0), s(s(0)), s(s(s(0))))$. $sum(0, 0, 0)$. $sum(0, s(s(0)), s(s(0)))$. $sum(s(0), 0, s(0))$. $sum(s(0), s(0), s(s(0)))$. $sum(0, s(0), s(0))$. $sum(s(s(0)), s(s(0)), s(s(s(s(0)))))$. $sum(s(s(0)), s(s(s(0))), s(s(s(s(0)))))$.

For this problem we consider four different hypothesis that cover completely the evidence. The theories are shown in Table 2.

To compute the $p(E|T)$ we employ an approach based on the proofs of the examples given T. Given a theory T, we give the same probability for each rule, then for every example of e, we traverse the corresponding SLD-tree of the proof taking into account the number of times a rule is applied. When we arrive to the root of the tree, we also consider if we still have to apply any substitution. For instance, if we have the program 4, the probability of example $sum(s(0), s(s(0)), s(s(s(0))))$ is $\frac{1}{32}$: we apply two rules once (i.e $\frac{1}{4}$), and then we still have to solve the substitution $X/s(s(0))$ ($\frac{1}{8}$). When we know every single probability of the elements of E, (if there are not repeated examples) then we can compute the probability of E using the method of Section 3.2.1.

In Table 3 we include the costs of coding these four theories with respect to the Evidence. We show the results of our approach, and the results of the $\mathcal{PC}$. As expected the theory that is more concise, it is cheaper to encode it, while the cost of $L(E|T)$ does not penalise the short theories. Note that with our approach, in the first case we don't need to code E since it is exactly T.

6 Conclusions

In this paper we have introduced a robust and well-founded coding for ILP, which allows us to code both the complexity of a program and the complexity of the evidence given the program, following a Bayesian interpretation of the MML principle. Our MML coding has been motivated by the frequently disregarded problems of previous (and old) codings in ILP: Model Complexity ($\mathcal{MC}$) will prefer very "patchy" programs to cope with infinite or sparse evidences; Proof Complexity ($\mathcal{PC}$) will prefer programs with simple proofs rather than simple programs. In the examples and discussion we introduced to illustrate this, we have seen that our coding can work well with sparse and dense evidences, and can also deal with noise. Hence enhances both earlier works on model complexity and proof complexity.

Our coding is general in the sense that it works for any logic program and positive evidence. Evidence is considered to possibly have repeated facts because this generalises the coding. Nonetheless, it is easy to have a more efficient coding scheme for evidences without repeated examples by adapting coding scheme 1 in Section 3.2. The coding we present here can be used to reconsider some of the earlier works on ILP systems and techniques which used simplicity, the MDL principle or the MML principle as a criterion for model selection or for ordering the search space. As future work, we plan to investigate two different extensions. The first (and easier, as mentioned above) is the application of the coding to probabilistic/stochastic logic programs. The second is the inclusion of negative evidence and the con-

1	$sum(0,0,0)$.	$sum(s(0),s(s(0)),s(s(s(0))))$.
	$sum(0,s(0),s(0))$.	$sum(s(0),s(0),s(s(0)))$.
	$sum(s(0),0,s(0))$.	$sum(s(s(0)),s(0),s(s(s(0))))$.
	$sum(0,s(s(0)),s(s(0)))$.	$sum(s(s(0)),s(s(0)),s(s(s(s(0)))))$.
2	$sum(0,X,X)$.	$sum(s(s(s(0))),s(0),s(s(s(s(0)))))$.
	$sum(0,s(X),s(X))$.	$sum(s(s(0)),s(0),s(s(s(0))))$.
	$sum(s(s(s(0))),0,s(s(s(0))))$.	
3	$sum(0,X,X)$.	$sum(0,s(X),s(X))$.
	$sum(0,s(s(X)),s(s(X)))$.	$sum(s(s(s(0))),0,s(s(s(0))))$.
4	$sum(0,X,X)$	$sum(s(X),Y,s(Z)) :- sum(X,Y,Z)$

Table 2: Theories for the sum problem.

							$\mathcal{EC}$		$\mathcal{PC}$	
T	#Rules	Lex.	Heads	Bodies	Vbles	L(T)	L(E|T)	L(T) + L(E|T)	L(E|T)	L(T) + L(E|T)
1	7.09	12.09	102.00	13.67	0.00	134.85	0.00	134.85	28.52	163.38
2	5.34	12.09	59.00	7.59	2.00	86.02	14.39	100.41	37.90	123.92
3	4.52	12.09	38.00	6.07	3.00	63.68	11.86	75.55	37	100.68
4	2.52	12.09	13.00	11.04	10.81	49.46	11.86	61.32	37	86.46

Table 3: Costs of the theories for the sum problem.

cept of “negative exception" in the MML coding, which will possibly involve the use of logic programs with negation.

References

[1] D. Conklin and I.H. Witten. Complexity-based induction. *Machine Learning*, 16(3):203–225, 1994.

[2] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

[3] D. L. Dowe, C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. MML for Inductive Logic Programming. Technical report, DSIC, UPV, 2007.

[4] M. Kovacic. Stochastic inductive logic programming, 1994.

[5] S. Muggleton. Learning from positive data. In *Inductive Logic Programming Workshop*, pages 358–376, 1996.

[6] S. Muggleton, A. Srinivasan, and M. Bain. Compression, significance and accuracy. In *Ninth Int. Conf. on Machine Learning*, pages 338–347, 1992.

[7] S. H. Muggleton. Inductive logic programming: Issues, results, and the challenge of learning language in logic. *Artificial Intelligence*, 114(1–2):283–296, 1999.

[8] S.H. Muggleton. Learning structure and parameters of stochastic logic programs. *Electronic Transactions in Artificial Intelligence*, 6, 2002.

[9] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

[10] J. Rissanen. Modelling by shortest data description. *Automatica*, 14:465–471, 1978.

[11] J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431, 1983.

[12] A. Srinivasan, S.H. Muggleton, and M. Bain. The justification of logical theories based on data compression. In *Machine Intelligence 13*, pages 87–121. 1994.

[13] C. S. Wallace. *Statistical and Inductive Inference by Minimum Message Length.* Ed. Springer-Verlag, 2005.

[14] C. S. Wallace and D. M. Boulton. An invariant Bayes method for point estimation. *Classification Society Bulletin*, 3(3):11–34, 1975.

[15] C. S. Wallace and D. L. Dowe. Minimum Message Length and Kolmogorov Complexity. *Computer Journal*, 42(4):270–283, 1999.

[16] Chris S. Wallace and D. M. Boulton. An information measure for classification. *Computer Journal*, 11(2):185–194, 1968.