Universität des Saarlandes
Max-Planck-Institut für Informatik
AG5

Learning Soft Inference Rules in Large and Uncertain Knowledge Bases

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Christina Teflioudi

angefertigt unter der Leitung von / supervised by

Dr. Martin Theobald

begutachtet von / reviewers

Dr. Martin Theobald

Prof. Dr. Gerhard Weikum

January 2011

Non-plagiarism Statement

I, Christina Teflioudi, hereby confirm that this thesis is my own work and that I have documented all sources used.

Signed: ______________________________

Date: ______________________________

Abstract

Recent progress in information extraction has enabled us to create large semantic knowledge bases with millions of RDF facts extracted from the Web. Nevertheless, the resulting knowledge bases are still incomplete or might contain inconsistencies, either because of the heuristic nature of the extraction process, or due to the varying reliability of the Web sources from which they were collected. One possible way of resolving both issues is to reinforce the knowledge base with deductive power by appending first-order logical inference rules, which help to describe and to further constrain the domain with which the ontology deals. In our work, we investigate learning these rules directly from the data using Inductive Logic Programming (ILP), a well known technique, which lies in the intersection of machine learning and logic. Although powerful, ILP is inherently expensive as there is a combinatorial growth of the search space when constructing these rules, as the size of the background knowledge grows. In addition, the evaluation of each rule becomes more expensive, as the number of the training examples is rising. Apart from that, it is not always obvious how to automatically select positive and negative training examples needed for learning new rules over an incomplete knowledge base. This Master thesis explores the issues involved when applying ILP in an incomplete and large RDF-knowledge base.

Acknowledgements

First and foremost, I am deeply grateful to my parents for their support. I would not be where I am today without their prayers, love and encouragement.

I am indebted to my advisor, Dr. Martin Theobald, for his guidance and support during my thesis.

I am also thankful to Prof. Gerhard Weikum and the members of the Database and Information Systems Group. They provided a motivating work environment. I have learned a great deal about conducting research by observing the members of the group.

I am thankful to my friends in Saarland University and in Greece for the psychological support they provided me.

Finally, I am grateful to Mrs. Jennifer Gerling and Mrs. Stephanie Jörg of the International Max Planck Research School for Computer Science for their help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem

Despite the recent progress in the field of information extraction in the web, its methods remain imperfect in terms of precision and recall. In addition to that, the sources, from which the facts are extracted, might contain contradictory or uncertain information and, of course, they might lack some information of interest. For these reasons, many information extraction tools provide, apart from the extracted fact itself, also a corresponding confidence value, which represents our degree of confidence that this specific fact is true. Such pairs of RDF facts and confidence values can then be used in order to populate an ontological knowledge base (e.g., YAGO [SKW07]). The resulting knowledge base still suffers from the uncertainty and incompleteness, which it inherited from the information extraction methods and sources used. The semantic web community proposes the use of first-order logic reasoning in order to tackle with these issues. In other words, we could append to our knowledge base first-order inference rules, which describe the domain, and hold also with some degree of confidence (soft rules). By using such rules, we can strengthen or weaken our confidence for the truth value of specific facts. Consider, for example, that we have the following two facts in our knowledge base:

$$bornIn(AlbertEinstein, Ulm)[0.7]$$

$$bornIn(AlbertEinstein, Munich)[0,8]$$

If we have a rule which states that a person is more likely to be born at the place where its parents used to live, like:

$$bornIn(z, y)\text{:-}hasChild(z, x), livesIn(z, y)[0.7]$$

and it happens to know that Einstein's parents lived in Ulm, then, immediately, our confidence for the corresponding fact gets stronger. Soft, inference rules can also be used as a possible remedy for incompleteness. Imagine, for example, that the user issues a query about the place, where Al Gore lives, and that the knowledge base itself contains no explicit fact that states this information. If we had a rule, according to which married people live together with their spouse, again with some confidence, like:

$$livesIn(x, y)\text{:-}isMarriedTo(x, z), livesIn(z, y)$$

and we know that Al Gore's wife, Tipper Gore, lives in Washington,D.C., then we can answer the query by inferring the fact $livesIn(AlGore, Washington, D.C.)$.

Rules, like the afore mentioned, are expressing situations that *usually*, but perhaps not always, hold in every day life and therefore are expected to be reflected in the data. This means that such rules can be learned directly from the facts included in the knowledge base itself, by using *Inductive Logic Programming (ILP)*, a technique which lies in the intersection of machine learning and logic.

Although ILP is powerful and well-studied, many issues arise when we apply ILP to a large and incomplete knowledge base. ILP is inherently expensive and its search space grows combinatorially with the size of the background knowledge. Therefore, the language bias should be carefully selected, such that it will impose an acceptable trade off between efficiency and expressivity. Apart from that, the evaluation of each rule becomes more expensive as the number of training examples is rising. A possible remedy for this problem is to use a sample of the original training examples, but the existing sampling methods for ILP are not taking into consideration the incompleteness and the absence of explicit negative examples, which are characteristic properties of our knowledge base. This fact makes these methods inappropriate for our setting and forces us to investigate the issue of sampling in incomplete knowledge bases. These last two issues, i.e., the incompleteness and the absence of negative examples, cause also problems in the way standard ILP measures the predictive power of the rules and, therefore, we had to introduce new measures for evaluating a rule's predictive power, which take incompleteness into consideration. Moreover, we introduced a measure for evaluating how general a clause is and in this case we force further refinement of this clause by taking advantage of the type information about the head predicate's arguments. In the end, we evaluated the rules produced by our rule learner in terms of their ability to produce new facts and of their ability to produce facts of high precision. Precision was assessed manually and with Wikipedia as ground truth.

This thesis was developed as a part of the URDF [TSSN10] project, a framework for efficient reasoning in uncertain RDF knowledge bases with soft and hard rules.

1.2 Contributions

We make the following contributions in this thesis:

- we implement an ILP-learner for mining soft, inference rules, in the form of Datalog clauses, from large and incomplete knowledge bases, which takes advantage of opportunities for early pruning of the hypothesis search space and tries to reduce the search space branching factor by applying a relation-preprocessing step.
- we introduce a measure of the predictive strength of the rules which takes into consideration the incompleteness of the knowledge base, in the sense of *missing entities* from a relation, and the absence of explicit negative examples
- we propose an improvement of the previous measure for the cases that our knowledge bases suffers also from incompleteness, in the sense of *missing facts* about entities which already exist in a relation
- we propose a heuristic for identifying and quantifying the incompleteness in this last case
- we introduce a measure for identifying overly general clauses and propose a way to construct useful clauses from clauses which are overly general by taking advantage of the type information about the arguments of the head predicate
- we investigate the issue of sampling in the case of an incomplete knowledge base and without explicit negative examples

1.3 Outline

This thesis is organized as follows: in Chapter 2 we introduce the basic terminology from logics, which we will need in order to present the essential background information on inductive logic programming (ILP). We also describe shortly basic notions and methods of semantic web. Chapter 3 presents the URDF framework for efficient reasoning in RDF knowledge bases with soft and hard rules, which was the main motivation for this thesis. There, we show how URDF enables query answering in a setting where the base data of automatically extracted facts exhibit incompleteness and inconsistency. Chapter 4 discusses the difficulties in applying ILP in a large and incomplete RDF knowledge base and introduces the main measures, which will be used in our approach. In Chapter 5 we combine all notions mentioned in Chapter 4 in order to present our implementation of an ILP rule-learner for large and incomplete knowledge bases. We give an experimental evaluation of the deductive power of the rules learned by our approach and a discussion of results in Chapter 6. Finally, in Chapter 7, we give the summary of this thesis and propose directions for future work.

Chapter 2

Related Work: Logic, Inductive Logic Programming and Semantic Web

In this chapter we will introduce the basic terminology for first order logic in Section 2.1, whereas in Section 2.2 we will describe the technique of Inductive Logic Programming. Section 2.3 discusses basic concepts of Semantic Web.

2.1 Logic Programming

In the following, we will briefly introduce the basic logic programming and deductive database terminology, as they are described in [KK], [LD93].

A first-order alphabet consists of variables, predicate symbols and function symbols. A *term* can be a *variable* or a *function symbol* immediately followed by a bracketed n-tuple of terms. For example, f(g(X),h) is a term when f, g, h are function symbols and X is a variable. A *constant* is a function symbol of arity 0, i.e., it is followed by a bracketed 0-tuple of terms, which means that it is also a term.

A predicate symbol immediately followed by a bracketed n-tuple of terms is called an *atomic formula* or *atom*. Whenever L is an atomic formula, both L and its negation $\neg L$ are *literals*. In this case, L is called a *positive literal*, and $\neg L$ is called a *negative literal*.

A *clause* is a formula of the form $\forall X_1 \forall X_2 ... \forall X_s (L_1 \vee L_2 ... \vee L_m)$ where each L_i is a literal and $X_1, X_2, \ldots, X_s$ are all the variables occurring in $L_1 \vee L_2 ... \vee L_m$. A clause can also be represented as a finite set of literals. Thus the set $\{L_1, L_2, \ldots, \neg L_i, \neg L_{(i+1)}, \ldots\}$ stands for the clause $(L_1 \vee L_2 \vee \ldots \vee \neg L_i \vee \neg L_{(i+1)} \vee \ldots)$ which is equivalently represented as $(L_1 \vee L_2 \vee \ldots \leftarrow L_i \wedge L_{(i+1)} \wedge \ldots)$. Most commonly this same clause is written as $(L_1, L_2, \ldots \leftarrow L_i, L_{(i+1)}, \ldots)$, where commas on the left hand side denote disjunctions, and commas on right hand side denote conjunctions. In the following we will use both the notation *:-* and $\leftarrow$ in order to denote the logical implication.

A *Horn Clause* is a clause which contains at most one positive literal. A *program clause* is a clause of the form $T \leftarrow L_1, L_2, \ldots, L_m$, where T is an atom and each of the $L_1, L_2, \ldots, L_m$ is of the form L or $\neg L$, where L is an atom. If a clause contains exactly one positive literal T, and it has the form $T \leftarrow L_1, L_2, \ldots, L_m$, where $T, L_1, L_2, \ldots, L_m$ are atoms, then it is called *definite program clause.* The positive literal T is the *head of the clause* and the conjunction of negative literals $L_1, L_2, \ldots, L_m$ is the *body of the clause.* A program clause with an empty body, i.e, a clause of the form $L \leftarrow$ is called a *positive unit clause.*

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, logic is used as a purely declarative representation language, which means that the program logic is expressed in terms of relations, represented as facts (literals without any variables) and rules (clauses). A computation is initiated by running a query over these relations. *Prolog* is a general purpose logic programming language. Relations in Prolog are defined by clauses. *Pure Prolog* is restricted to Horn clauses.

If a program clause contains no function symbols of non-zero arity, it is called a *Datalog clause*, which is syntactically a subset of Prolog. In other words, in Datalog clauses only variables and constants can be used as predicate arguments. A program clause can be *non-recursive* if the predicate symbol in the head does not appear in any of the literals in the body. Otherwise, it is a recursive program clause. If the arguments of the predicates have types, then the program clause is *typed.* A *typed* program clause of the form $T \leftarrow L_1, L_2, \ldots, L_m$ is called a *database clause.*

A *deductive database* (DDB) is a set of database clauses. A set of definite program clauses is called a *definite logic program* and a set of program clauses with the same predicate symbol and arity (number of arguments) in their head is a *predicate definition.* A set of clauses, in general, is called a *clausal theory* and represents the conjunction of its clauses.

Literals, clauses and clausal theories are all *well formed formulae* (wff). If E is a wff or term and $vars(E)$ denote the set of variables in E, then E is said to be ground if and only if $vars(E) = \emptyset$.

2.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a technology combining the principles of inductive machine learning with the representation of logic programming, and it aims at inducing general rules starting from specific observations and background knowledge. By specific observations we mean training or example data in the form of grounded facts. i.e., literals without any variables. The notion of background knowledge and its role in learning will be explained in Section 2.2.2. In the following, we will introduce the basic techniques used in ILP as described in [LD93].

2.2.1 Problem Statement in Classical ILP

To define the problem of inductive concept learning, one first needs to define a concept. According to [LD93], if U is a universal set of objects (or observations), a concept C can be formalized as a subset of objects in $U : C \subseteq U$. For example, U may be the set of all patients in a register and C the set of all patients having a particular disease. Learning a concept C means to recognize objects in C, i.e., being able to tell whether $x \in C$, for each $x \in U$.

From now on, we will use the term *fact* for the description of an object (or observation) and *hypothesis* for the description of the concept to be learned. An example e for learning a concept C is a labeled fact, with label + if the object is an instance of the concept C, and with label - otherwise. Let E denote a set of examples, from which E^+ is the set of positive examples (with label +) and E^- is the set of negative examples (with label -).

Concepts and objects need to be described in some *description language* L. Usually, we describe objects extensionally, i.e, in the form of ground facts, and concepts intensionally, i.e., in the form of rules, which allows for a more compact representation of the concept. Various logical formalisms have been used in inductive logic programming systems to represent examples and concept descriptions, ranging from propositional logic (propositional learners) to full first order predicate calculus (relation learners). In relation learners, the languages used to describe examples, background knowledge and concepts are typically subsets of first-order logic, with most popular language the subset of Horn clauses.

If the description of the object satisfies the description of the concept, we will say that the concept description *covers* the object description, or that the object description *is covered by* the concept description.

In the so-called *single concept learning*, a concept description is induced from facts labeled + and -, i.e., from examples and counterexamples of a single concept C. The problem of learning a single concept C can be stated as follows:

Given a set E of positive and negative examples of a concept C, find a hypothesis H, expressed in a given concept description language L, such that:

- *every positive example is covered H*
- *no negative example is covered by H*

To test the coverage of our hypothesis, we need to introduce the function $covers(H, e)$, which returns the value *true* if e is covered by H, and $false$ otherwise. In logic programming, where a hypothesis is a set of program clauses and an example is a ground fact, forward chaining procedure or a form of SLD-resolution rule of inference [KK] can be used to check whether e is *entailed* by H. In other words, e is covered by H given the background knowledge B, if e is a logical consequence of $B \cup H$.

The function $covers(H, e)$ can be extended to work for sets of examples in the following way:

$$covers(H, E) = \{e \in E | covers(H, e) = true\} \tag{2.1}$$

returning the set of facts from E, which are covered by H.

In the problem statement of inductive concept learning, it is required that the hypothesis H covers all the positive and no negative examples. In this case, we say that the hypothesis is *complete* (covers all positive examples) and *consistent* (covers no negative example).

2.2.2 The Role of the Background Knowledge

Concept learning can be viewed as searching the space of concept descriptions. If the learner has no prior knowledge about the learning problem, it learns exclusively from examples. However, difficult learning problems typically require a substantial body of prior knowledge, which we will call the background knowledge. The hypothesis language L and indirectly the background knowledge B determine the search space of possible concept descriptions (hypothesis space). Therefore, the background knowledge has to be included in the problem statement of inductive concept learning:

Training Examples	**Background Knowledge**
daughter(mary,ann). +	parent(ann,mary).
daughter(eve,tom). +	parent(ann, tom).
daughter(tom,ann). -	parent(tom,eve).
daughter(eve,ann). -	parent(tom,ian).
	female(ann).
	female(mary).
	female(eve).

TABLE 2.1: Learning the daughter(X,Y) relation

Given a set E of positive and negative examples of a concept C and background knowledge B, find a hypothesis H, expressed in a given concept description language L, such that H is complete and consistent with respect to B and E. For this definition we need to extend again the notion of the covering function as:

$$covers(B, H, E) = covers(B \cup H, E) \tag{2.2}$$

2.2.3 An Example ILP Problem

Let us illustrate the ILP task on a simple problem of learning family relations. The task in this example is to define the target relation $daughter(X, Y)$, which states that the person X is a daughter of person Y in terms of the background knowledge relations female and parent. In Table 2.1 we can see the examples and the background knowledge.

In the hypothesis language of Horn clauses it is possible to formulate the following definition of the target relation:

$$daughter(X, Y) \leftarrow female(X), parent(Y, X).$$

This definition is complete and consistent with respect to the background knowledge and the training examples. Depending on the background knowledge, the language L and the complexity of the target concept, the target predicate definition may consist of a set of clauses, such as:

$$daughter(X, Y) \leftarrow female(X), father(Y, X).$$

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$

2.2.4 Dimensions of ILP

Inductive logic programming systems can be divided along several dimensions:

- They can learn either a single concept or multiple concepts. In the latter case, labels denote different concept names representing different classes. Then, the set of training examples E can be divided into subsets of positive and negative examples for each individual concept-class.
- They may require all the training examples to be given before the learning process (batch learners), or they may accept examples one by one (incremental learners).
- They may be interactive or non-interactive. In an interactive system, during the learning process, the learner may rely on an oracle to verify the validity of generalizations generated by the learner.
- They may try to learn a concept from scratch, or they can accept an initial hypothesis (theory), which is revised in the learning process (theory revisors).

Typically, ILP systems are either batch, non-interactive systems that learn a single predicate from scratch and are called empirical ILP systems (such as GOLEM [Plo71], FOIL[Qui90], MFOIL [Dˇ93] and LINUS[DL91]), or they are interactive and incremental theory revisors that learn multiple predicates, also called interactive or incremental ILP systems. Empirical ILP systems are able to learn concepts from large collections of examples, and for this reason we will concentrate on them.

2.2.5 Structuring the Hypothesis Space

Concept learning can be viewed as a search problem [Mit82]. States in the search space (or hypothesis space) are concept descriptions, and the goal is to find one or more states satisfying some quality criterion. A learner can be described in terms of its search space, its search strategy, and search heuristics.

In ILP, the search space is determined by the language of logic programs L consisting of the possible program clauses of the form $T \leftarrow Q$, where T is an atom $p(X_1, X_2, ..., X_n)$ and Q is a conjunction of literals $L_1, L_2, ..., L_m$. The vocabulary of predicate symbols q_i in the literals L_i of the body of a clause is determined by the predicates from the background knowledge B.

In order to search the space of program clauses systematically, it is useful to structure the space by introducing a partial ordering into a set of clauses based on the so-called θ-subsumption. To define θ-subsumption, we first need to define substitution:

A substitution $\theta = \{X_1/t_1, ..., X_k/t_k\}$ is a function from variables to terms. The application $W\theta$ of a substitution θ to a well formed formula (wff) W is obtained by replacing all occurrences of each variable X_i in W by the same term t_j. [KK]

Let c and c' be two program clauses. Clause c θ-subsumes c' if there exists a substitution θ, such that $c\theta \subseteq c'$, where the clauses c, c' are represented as finite sets of literals, as described in Section 2.1.

For example, let $c = daughter(X, Y) \leftarrow parent(Y, X)$. Then c θ-subsumes the clause $c' = daughter(X, Y) \leftarrow female(X), parent(Y, X)$ under the empty substitution $\theta = \emptyset$, since the set

$$\{daughter(X, Y), \neg parent(Y, X)\}$$

is a proper subset of

$$\{daughter(X, Y), \neg female(X), \neg parent(Y, X)\}\,.$$

θ-subsumption imposes a partial ordering between the clauses, which can be represented in the form of a *subsumption lattice*. Having structured the search space, the learner can search it blindly, in an uninformed way, or heuristically. In uninformed search, the depth-first or breadth-first search strategy can be applied. On the other hand, in heuristic search, the usual search strategies are hill-climbing or beam search (see Section 2.2.7).

θ-subsumption introduces also the syntactic notion of generality. Clause c is at least as general as clause c' ($c \leq c'$) if c θ-subsumes c'. We say that c' is a specialization (refinement) of c and that c is a generalization of c'.

There is an important property of θ-subsumption: if c θ-subsumes c', then c logically entails c' (c' is a logical consequent of c). The reverse is not always true.

We can use the above property of θ-subsumption in order to prune large parts of the search space [MdR94]:

- When generalizing from c to c', all the examples covered by c will also be covered by c'. So, if c is inconsistent (covers a negative example), then all its generalizations will be inconsistent too. Hence, the generalizations of c do not need to be considered.
- When specializing c to c', an example not covered by c will not be covered also by c'. So, if c does not cover a positive example none of its specializations will. Hence, the specializations of c do not need to be considered.

θ-subsumption provides the basis for two important ILP techniques:

- Bottom-up building of least general generalization from training examples, relative to background knowledge, and
- Top-down searching of refinement graphs.

Generalization techniques search the hypothesis space in a bottom-up manner. They start from the training examples (most specific hypotheses) and search the hypothesis space by using generalization operators. Generalization techniques are best suited for interactive and incremental learning from few examples. On the other hand, specialization techniques search the hypothesis space top-down, from the most general to specific concept descriptions, using specialization operators. Specialization techniques are better suited for empirical learning. Therefore, in the following we will concentrate on specialization techniques.

2.2.6 Specialization Techniques

Specialization techniques search the hypothesis space using a θ-subsumption based specialization operator (or refinement operator). Given a language bias L, a refinement operator ρ maps a clause c to a set of clauses $\rho(c)$ which are specializations (refinements) of c:

$$\rho(c) = \left\{c' | c' \in L, c < c'\right\}$$

A refinement operator typically computes only the set of minimal (most general) specializations of a clause under θ-subsumption. It employs two basic syntactic operations on a clause:

- Apply a substitution on the clause, and
- add a literal to the body of the clause.

Top-down learners start from the most general clauses and repeatedly refine them until they no longer cover any negative example; during the search they ensure that the clauses considered cover at least one positive example.

2.2.7 A Unifying Framework for Specialization

In this section, we will present a generic top-down empirical ILP algorithm [LD93] which builds a unifying framework for most of the empirical ILP systems. But first we need to establish some notation:

A hypothesis H is a set of program clauses $\{c' | c' \in L\}$. The set E is the given (initial) training set. It consists of positive examples E^+ and negative examples E^-, $E = E^+ \cup E^-$. Let n denote the number of examples in E, n^+ of which are positive and n^- are negative. The hypothesis H currently built by the learner is called the current hypothesis, and the currently built clause c the current clause. The current training set E_{cur} is the set of training examples given to the learner when building the current clause c. The local training set E_c is a subset of the current training set E_{cur} of examples covered by c. Let $n(c)$ denote the number of examples in the local training set, $n^+(c)$ of which are positive and $n^-(c)$ are negative.

The hypothesis construction (Algorithm 2.1) assumes the current training set E_{cur} (initially set to the entire training set E) and the current hypothesis H (initially set to the empty set of clauses). The algorithm consists of two repeat-loops, referred to as the covering loop and the specialization loop. The covering loop performs hypothesis construction and the specialization loop performs clause construction.

Algorithm 2.1: Generic top-down specialization ILP algorithm

```
1  Initialize $E_{cur} := E$
2  Initialize $H := \emptyset$
3  repeat {Covering}
4      Initialize $c : T \leftarrow$
5      repeat {Specialization}
6          find the best refinement $c_{best} \in \rho(c)$
7          assign $c := c_{best}$
8      until Necessity Stopping Criterion is satisfied
9      Add c to H to get the new hypothesis $H' := H \cup \{c\}$
10     Remove positive examples covered by c from $E_{cur}$ to get new training set
       $E'_{cur} := E_{cur} - covers(B, H', E^+_{cur})$.
11     $E_{cur} := E'_{cur}$
12     $H := H'$
13 until Sufficiency Stopping Criterion is satisfied
14
   Output: Hypothesis H
```

The covering loop constructs a hypothesis in three main steps:

1. Construct a clause,
2. add the clause to the current hypothesis, and
3. remove from the current training set the positive examples covered by the clause

This loop is typically repeated until all the positive examples are covered.

The main part is clause construction (step 1) implemented in the specialization loop. Clause construction is performed by means of the refinement operator ρ, which takes the current clause c and builds a set of its refinements $\rho(c) = c'$ by adding a literal L to the body of c (initially c is a clause with empty body). In hill-climbing-based search of the refinement graph, the algorithm keeps one best clause and replaces it with its best refinement in each specialization step, until the necessity stopping criterion is satisfied. In beam search, the algorithm keeps several best clauses instead of just one.

The two repeat loops are controlled by two stopping criteria:

- In the specialization loop, the necessity stopping criterion decides when to stop adding literals to a clause.
- In the covering loop, the sufficiency stopping criterion decides when to stop adding clauses to the hypothesis.

In domains with perfect data, the necessity criterion requires consistency (no covered negative examples) and the sufficiency criterion requires completeness (all positive examples covered). In domains with imperfect data, heuristic stopping criteria are applied. The simplest heuristic is the expected accuracy of a clause $A(c) = P(e \in E^+|c)$. The expected accuracy is defined as the probability that an example covered by the clause c is positive. Different probability estimates can be used to compute $P(e \in E^+|c)$, the most frequent one being the relative frequency of covered positive examples:

$$P(e \in E^+|c) = \frac{n^+(c)}{n^+(c) + n^-(c)} \tag{2.3}$$

2.2.8 The Complexity of Inductive Logic Programming

Gottlob et al. in [GLS99] investigated the complexity of ILP, assuming that H may contain no more than k literals, where k defines the size of the problem instance. It turned out that the problem is not only NP-hard, but resides in Σ_2^P, i.e., at the second level of the polynomial hierarchy of problem complexity. Less formally, Gottlob et al. showed that two sources of exponential complexity are intrinsic to the ILP problem:

1. One rests in the size of the search space that must be explored to find a clause that solves the ILP problem (which depends on the branching factor of the search, i.e., how many refinements are generated from a clause). In Sections 4.6, 5.3 and 5.1 we discuss some ideas about reducing the branching factor.

2. Another one is incurred by the subsumption test, which is NP-complete, and which must be conducted for each clause explored (i.e., the fitness test of each explored clause with respect to the data by checking for each example whether or not it is subsumed by the clause). In Section 5.2 we will see how we can keep the number of subsumption tests low in our algorithm.

2.2.9 Existing ILP Systems and Applications

In this section, we will give a list and short description of some existing learning systems. The following list is far from complete but includes some of the most popular and interesting ILP systems.

- **Progol** generates several clauses stepwise. If examples are covered by the current set of clauses, they are removed. This process continues until sufficiently many examples have been covered. For finding an appropriate clause, Progol selects a seed example and builds a most specific clause (or bottom clause) using the technique of inverse entailment [MBR95]. Then this clause is generalized in the learning process.
- **Aleph**[1] (A Learning Engine for Proposing Hypotheses) is one of the most popular ILP systems and the successor of P-Progol (the prolog implementation of Progol). It implements several algorithms and ideas in the field of ILP and can actually simulate the behavior of other learning systems.
- **Golem** is based on the idea of relative least general generalizations (RLGGS) [Plo71]. Like Progol, it generates several clauses stepwise, but for the learning of a single clause the idea of RLGGS is employed. The RLGGS is a least general hypothesis relative to background knowledge with respect to given (positive) examples. Golem first computes the RLGGS of randomly selected pairs of examples. The best result, i.e. the hypothesis with the best coverage of those random pairs' RLGGS, is picked. In the next loop, the system computes the RLGGS of the currently best hypothesis and randomly selected examples. Again, the best result is chosen and the system continues until a further loop does not increase coverage of the hypothesis. Details are described in [MF90].
- **FOIL** uses background knowledge in extensional form, i.e., relational/grounded tuples are used and the target language are function free Horn clauses. In FOIL the induction of a single clause starts with an empty clause body. The clause body is iteratively specialized by adding literals. The literals are chosen by placing

[1]http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph_toc.html

variables to their appropriate argument places, which need to be specified before. To choose a literal amongst the possible candidates, FOIL uses the information gain heuristic, which is based on how much the literal helps in distinguishing between positive and negative examples [QCj93].

- **nFOIL** [LKR] is a system which tries to integrate naive Bayes and FOIL in a probabilistic ILP system. nFOIL employs naive Bayes as a criterion to guide the search. In nFOIL the coverage function is probabilistic; we are not counting the examples covered by some clause, but we calculate the probability that the clause covers these examples. In general, probabilistic inductive logic programming[RK08] received lots of interest lately. In the system **ProbFOIL** [RT10] both the examples themselves as well as their classification can be probabilistic.

ILP learning systems have been applied in several scenarios like:

- Learning drug structure activity rules. In this scenario the aim is to understand the relationship between chemical structure and activity. ILP systems are used to learn rules relating the structure to an activity and can be used to find promising chemical structures, i.e., to increase the success rate of pharmaceutical companies [KSS95].
- Natural Language Processing provides several application areas for inductive reasoning: one application area is part of speech tagging. Inductive reasoning has been applied for several languages, including English [Cus96]. ILP has also been used to extract relations from text [HPRW10], which is useful for building knowledge bases from text corpora.
- Robot Discovery is a research area, which aims to find out to which extent robots can learn about their environment (for example learning the concepts of "movability" or "obstacle") by conducting experiments and collecting data [LaB08].
- Detection of traffic problems as an ILP application area has been explored in [DJM$^+$98] with the goal of identifying problematic areas with respect to traffic jams and accidents, using the ILP tools Claudien and Tilde. Those tools could identify most critical sections correctly and learn plausible rules.
- For the Semantic Web: in [LA] the authors focus on learning OWL class expressions in Description Logics by developing the Protege[2] plug-in DL-Learner[3], whereas in [SEWD10] the authors are trying to learn first-order horn clauses from web text.

[2]http://protege.stanford.edu/
[3]http://aksw.org/Projects/DLLearner

2.3 Semantic Web, RDF/S, OWL and Description Logics

Semantic Web is a group of methods and technologies to allow machines to understand the meaning - or "semantics" - of information on the World Wide Web. The term was first used by World Wide Web Consortium (W3C) director Tim Berners-Lee. He defines the Semantic Web as "a web of data that can be processed directly and indirectly by machines". In other words, semantic web aims to make web resources, i.e., data and services, more readily accessible to automated processes.

The term "Semantic Web" is mainly used to describe the model and technologies proposed by the W3C. These technologies include the Resource Description Framework (RDF), a variety of data formats (e.g., RDF/XML, N3, Turtle, N-Triples), and constraints such as RDF Schema (RDFS) and the Web Ontology Language (OWL), all of which are intended to provide a formal description of concepts, terms, and relationships within a given knowledge domain. In Figure 2.1[4] we can see the W3C standards that have emerged in the form of a layer cake.

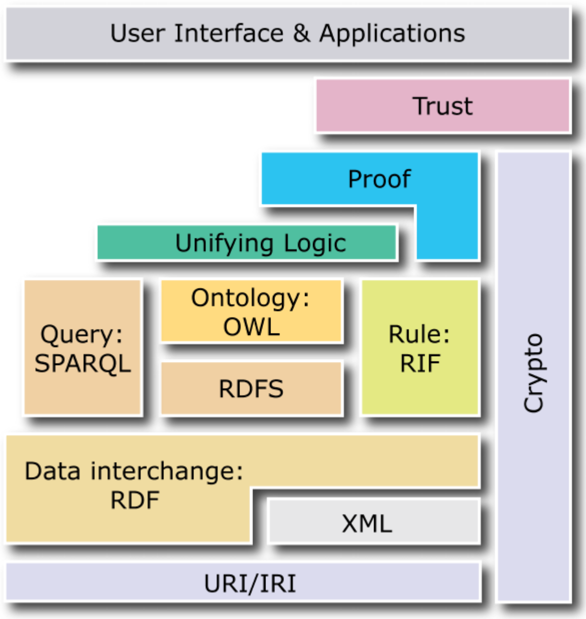


FIGURE 2.1: Semantic Web Layer Cake

The Resource Description Framework (RDF) is a W3C standard for expressing statements about resources. According to W3C[5]:

RDF is a standard model for data interchange on the Web. It extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a "triple"). Using this simple model,

[4]Source: http://www.w3.org/2007/03/layerCake.png
[5]http://www.w3.org/RDF/

it allows structured and semi-structured data to be mixed, exposed, and shared across different applications.

This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations.

Statements in RDF are stored as subject-predicate-object triples, which together form a labeled directed graph. Conceptually, the subject corresponds to an entity, the predicate to a property of the entity and the object to a value of the property. In Figure 2.2[6], a resource Eric Miller is described as a person with the specified name, email address, and title.

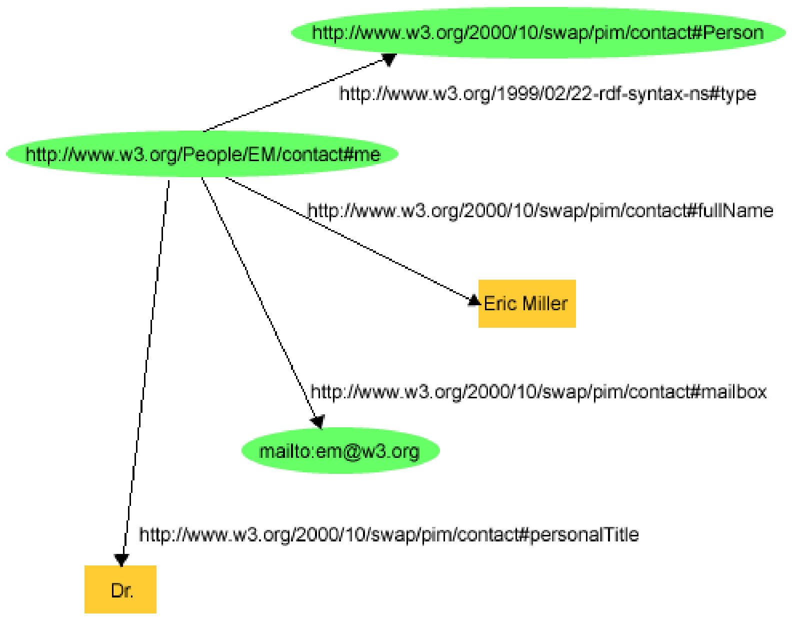

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/People/EM/contact#me
http://www.w3.org/2000/10/swap/pim/contact#fullName
Eric Miller
http://www.w3.org/2000/10/swap/pim/contact#mailbox
mailto:em@w3.org
http://www.w3.org/2000/10/swap/pim/contact#personalTitle
Dr.

FIGURE 2.2: RDF graph describing Eric Miller

RDF Schema[7] (RDFS) is a language designed to create RDF vocabularies, and it is practically a subset of the OWL ontology language. OWL (Web Ontology Language) is in essence based on description logics (see below) extended by several features to make it suitable as a web ontology language, e.g. using URIs/IRIs as identifiers, imports of other ontologies, etc, and it comes in three flavors: OWL Lite, OWL DL, and OWL Full. OWL Full contains features not expressible in description logics, but needed to be compatible with RDFS, i.e., OWL Full can be seen as the union of RDFS and OWL DL.

[6]Source: RDF Primer (http://www.w3.org/TR/rdf-primer/)
[7]http://www.w3.org/TR/rdf-schema/

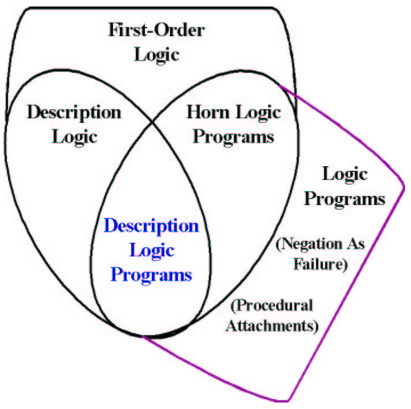


FIGURE 2.3: Expressive overlap of DL, Horn clauses and logic programming

Description logics (DL) is the name of a family of knowledge representation (KR) formalisms and they can essentially be understood as fragments of first-order predicate logic. They have less expressive power, but usually decidable inference problems and a user-friendly variable free syntax. Description logics represent knowledge in terms of objects, concepts, and roles. Concepts formally describe notions in an application domain, e.g., one could define the concept of being a father as “a man having a child”. Objects are members of concepts in the application domain and roles are binary relations between objects. Objects correspond to constants, concepts to unary predicates, and roles to binary predicates in first-order logic. In description logic systems, information is stored in a knowledge base, and it is sometimes divided in two parts: TBox and ABox. The ABox contains assertions about objects. In other words, it relates objects to concepts and other objects via roles. The TBox, on the other hand, describes the terminology by relating concepts and roles.

The semantic web standard for representing rules is RuleML. RuleML is based on Datalog restricted to unary and binary predicates. Datalog and description logics, however, have different expressive powers. For example, cardinality restrictions can be expressed in DL, but not in Datalog. On the other hand, Datalog belongs to logic programming clauses and, therefore, it can express for example recursion, which is not possible in first order logic, in general. Nevertheless, Datalog and DL are overlaping, i.e., there are statements that can be expressed in both of them, e.g. the transitivity property ($P \sqsubseteq^+ P$ in DL and $P(x,z) \leftarrow P(x,y), P(y,z)$ in Datalog). In [GHVD03] the authors discuss the expressive power of description logics and of logic programming clauses (like Datalog). Figure 2.3, taken from [GHVD03], shows the overlap between DL, Horn clauses and logic programming, part of which is also Datalog.

The final aim for using OWL is to formalize a domain by defining classes and properties about these classes, to define individuals and assert properties about them, and to reason about these classes and individuals in order to derive logical consequences, i.e, facts not literally present in the ontology, but entailed by the semantics. The last of the afore mentioned goals of OWL is also the motivation for our work, i.e., to perform inference in order to tackle the incompleteness and uncertainty which exist inherently in knowledge bases extracted from web sources. (The reasoning framework which we used will be described in the next chapter). In our case, we are interested in learning soft rules, i.e., which *usually* hold, in the form of well-formed Datalog clauses. These rules should describe definitions and connections between roles (binary predicates) of a knowledge base containing facts automatically extracted from the web, i.e., a knowledge base which is large and incomplete.

Chapter 3

URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules

This chapter presents the URDF framework for efficient reasoning in RDF knowledge bases with soft and hard rules, as described in [TSSN10]. Work on this thesis developed from the need to learn the soft inference rules directly from the data. Section 3.1 discusses the problem that URDF is trying to solve, which is query answering over large knowledge bases which exhibit inconsistency and/or incompleteness. Section 3.2 presents the representation model of URDF. In Section 3.3 the reasoning model of URDF is described.

3.1 Motivation

Even though there was lots of progress lately in the field of information extraction, its methods are inherently incomplete, which consequently means that the resulting extracted knowledge bases are still incomplete and/or inconsistent. By incompleteness, we mean the inability to capture all information related to the domain we are interested in, whereas inconsistency arises when the extracted information violates some properties of this domain. The reasons for incomplete or inconsistent information in knowledge bases might include the fact that the sources, from which information is extracted, usually do not contain complete information, or that they contain incorrect or contradicting information. Apart from that, information extraction tools and methods might not be able to capture all information in the sources or might not be able to handle how the information is presented, thus, leading to incompleteness and inconsistency.

The goal of URDF is to efficiently answer queries over potentially incomplete and inconsistent knowledge bases. Incompleteness might lead to empty results, whereas inconsistency leads to contradicting answers or derivation of answers from contradicting data. Starting with a knowledge base containing extracted facts with some confidence values assigned to them, URDF uses soft rules, which describe relations between facts in the corresponding domain, in order to generate more facts and extend the initial knowledge base. In this way, it tries to overcome incompleteness. In order to detect inconsistency, hard rules describing integrity constraints are added to the knowledge base. When querying a URDF knowledge base, multiple weighted answers will be returned, which taken together can violate these hard rules. Therefore, URDF uses an approximate weighted MAX-SAT solver. Usual weighted MAX-SAT solvers return a truth assignment for the facts such that the sum of the confidences of the facts that have value true is maximized, but they do not take into consideration hard rules, which should always hold. Unlike the usual MAX-SAT solvers, the MAX-SAT solver, which URDF uses, ensures that the answers it produces do not violate any integrity constraints. Figure 3.1 depicts the different components of URDF. UDRF uses the facts in the knowledge base and the rules (both soft and hard) together with a query in order to produce a dependency graph for this query (grounding phase). How exactly the dependency graph is created will be discussed in Section 3.3. Then, the dependency graph is given as input to the MAX-SAT solver, which tries to find a truth a assignment to all ground clauses such that the sum of the weights of the satisfied clauses is maximized, but at the same time without violating any of the hard rules.

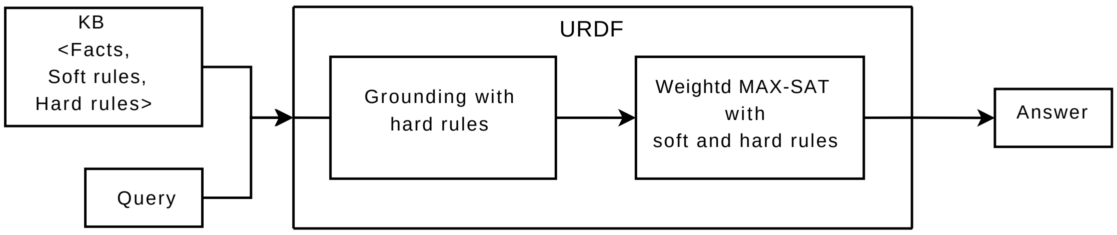


FIGURE 3.1: URDF

3.2 Representation Model

A **knowledge base** in URDF is defined as a triple $KB = \langle \mathcal{F}, \mathcal{C}, \mathcal{S} \rangle$ consisting of RDF base facts, $\mathcal{F}$, a set of soft clauses $\mathcal{C}$, and a set of hard rules (or integrity constraints) $\mathcal{S}$.

An RDF graph (see section 2.3) is a directed, labeled multi-graph, in which nodes are entities and labeled edges represent relationships between the entities.

3.2.1 Facts

The set $\mathcal{F}$ of facts in a URDF knowledge base contains RDF facts. Facts express binary relationships between entities. For the following, we will use prefix notation, e.g., $r(x, y)$ for a triplet $x - r - y$. Each fact has a confidence value, which reflects how reliable the fact is. The higher this confidence value is, the more reliable the fact. The assignment of weights to facts can depend on factors like the reliability of the source, from which a fact was extracted, the format in which a source presents facts, and the techniques used for extraction.

3.2.2 Soft Rules

Soft rules are used to generate new facts, which will compensate the incompleteness problem. They correspond to well-formed Datalog rules with weights. An example of a soft rule is:

$$livesIn(x, y) \leftarrow isMarriedTo(x, z) \wedge livesIn(z, y)[0.4]$$

According to this rule, a person lives in the same place as his or her spouse. A soft rule has a non-negative real valued weight. The higher the weight, the higher the confidence of the derived fact is. We allow only Horn rules (see Section 2.1), namely rules where at most one literal is positive. Note that in a Datalog representation only simple literals with no nested predicates or function calls are allowed in the rules.

These rules can either be handcrafted by some domain expert, or they can be learned directly from the data using machine learning techniques. This thesis focuses on how to learn such rules from the data using Inductive Logic Programming (see Section 2.2).

3.2.3 Hard Rules

Hard rules are a distinct set of rules which define mutually exclusive sets of facts and are used to enforce integrity constraints. If a *possible world* or *Herbrand interpretation* assigns a truth value to each possible ground atom in the knowledge base, then we can define a ground hard rule in the following way:

Definition 3.1 (Truth of a Ground Hard Rule)**.** A ground hard rule is true in a possible world, if at most one fact is true in the hard rule.

An example of a ground hard rule is

$$\{bornIn(Al\_Gore, USA), bornIn(Al\_Gore, Italy), bornIn(Al\_Gore, Spain)\}.$$

In the above example, the hard rule expresses a functional constraint on the predicate $bornIn$, i.e., Al Gore can only be born in one place.

In other words, a hard rule is a special kind of rule, which has no weight and it should always be satisfied. Therefore, it partitions the knowledge base in such a way, so it can always be satisfied. For the above example, the hard rule expressing the functional constraint of the $bornIn$ predicate partitions the knowledge base in three parts, in each of which only one of the three facts can be true and the other two should be false.

3.3 Reasoning Model

The URDF reasoning framework combines classic Datalog-style first-order reasoning with a generalized weighted Max-Sat solver for both soft and hard rules. In other words, given a query in the form of a set of non-grounded atoms, we aim to find a truth assignment to the grounded query atoms and to the grounded atoms of the dependency graph of this query (i.e., the facts that are relevant for answering the query), such that the sum of the weights over the satisfied soft rules is maximized, without violating any of the hard constraints. The definition of dependency graph follows:

Definition 3.2 (Dependency Graph)**.** Given a $\mathcal{KB} = \langle \mathcal{F}, \mathcal{S}, \mathcal{C} \rangle$ and a conjunctive query Q, then:

- All possible groundings $f \in F$ of the query atoms $q \in Q$ are facts in the dependency graph D.
- If a grounded fact f_n is in D, then all grounded facts $f_1, ..., f_n - 1$ of all grounded soft rules $C \in \mathcal{C}$, in which f_n is the head, are also in D.
- If a grounded fact f is in D, then all grounded facts $f_1, ..., f_k$ of the grounded hard rule $S \in \mathcal{S}$, which are mutually exclusive to f, are also in D.

Notice that the above definition already yields a recursive algorithm to compute the dependency graph, called SLD resolution [AvE82] in Prolog and Datalog.

URDF allows also for the formulation of recursive rules (the same predicate occurs both in the head and the body of the rule) as well as mutually recursive sets of rules (one rules produces grounded facts as input to another rule and vice versa).

The final answer to a query is generated by the special, weighted MAX-SAT solver. The MAX-SAT solver takes as input the dependency graph generated by the reasoner and tries to find an assignment of truth values to ground clauses such that the sum of the weights of the satisfied clauses is maximized, but at the same time without violating any of the hard rules expressed in the form of competitor sets (Hard Rules).

Chapter 4

ILP for Large and Incomplete RDF Knowledge Bases

In this chapter we will discuss the basic problems that arise when applying ILP in a large and incomplete knowledge base. We will define the measures of confidence, support and speciality ratio, which will guide our approach as described in Sections 5.1 and 5.4.

4.1 Motivation

As we have already seen, we are trying to learn rules in the form of Horn clauses of the form:

$$\overbrace{target(x_1, ...)}^{head} \text{:-} \overbrace{L_1(x_1, ...), L_2(x_j, ...), \ldots, L_m(x_k...)}^{Body} \tag{4.1}$$

In the case of an RDF knowledge base, all examples are of the form $target(x, y)$ where x is the subject and y the object, and $target$ is the RDF property between x and y for which the learning is done. In such a setting a positive example can be seen as a pair (x, y), which makes the formula $target(x, y)$ evaluate to true. On the other hand, a negative example can be also seen as a pair (x, y), which makes the formula $target(x, y)$ evaluate to false. Our rules are well-formed Datalog formulas, which means that for the construction of a rule, we require that both variables in the head should also appear in the body. This means that the body can also be seen as a mechanism, which takes as input x (or y) and predicts the value (or values) of y (or x) which make the formula $target(x, y)$ evaluate to true. In the following, we will see the target relation (or head) and the body of the rule as sets of pairs (x, y).

Remember that our initial motivation for learning rules was query answering. In the following, we will distinguish three cases:

- Case 1: we assume that we are given as input the variable x and we are trying to learn a rule (for a specific target predicate) that predicts the correct value(s) for y. In other words, we are simulating the case, in which the user is interested in a specific relation and subject and asks about the object (e.g. $livesIn(AlGore, ?y)$).
- Case 2: we assume that we are given as input the variable y and we are trying to learn a rule (for a specific target predicate) that predicts the correct value(s) for x. In other words, we are simulating the case, in which the user is interested in a specific relation and object and asks about the subject
(e.g. $familyNameOf(?x, AlbertEinstein)$).
- Case 3: we assume that we are given as input both the variables x and y and we are just trying to learn a rule (for a specific target predicate) that predicts if the fact $target(x, y)$ is true or false.

The idea of having an input (or focal) argument, which is expected to be bound during a query, is also used in [CKM+05], although there this idea is used in an algorithm for automatically learning the arguments' modes, as it will be explained in Section 5.1

Moreover, note that in our setting, positive examples can be translated into facts about the target relation. That is, we do not have positive examples other than the facts in our knowledge base. Note also, that at the same time the knowledge base can be incomplete, which means that there are many other pairs (x, y) that do not exist in it and can make the formula $target(x, y)$ evaluate to true, which makes a closed world assumption not suitable for our setting. Similarly, a knowledge base usually contains only facts that are true, which means that we also lack the presence of explicit negative examples.

In the following, we will address these issues in more detail and we will describe the basic measures by which our algorithms and implementation are driven.

4.2 Confidence

As we have already seen in the previous chapters, the standard ILP setting requires the exploration of the search space until a solution, which covers all positive (completeness) and no negative examples (consistency), is found. If the domain includes some noise, then the requirements for consistency and completeness need to be relaxed. In this case, the more convenient measure of accuracy (see section 2.2.7) is introduced. That is, the search stops if a solution, which has accuracy above some threshold, is found. The *expected classification accuracy* of a clause c: *head*:-*body* is defined as:

$$A(c) = P[e \in E^+|c] = P[head|body] = \frac{P[head \wedge body]}{P[body]} \tag{4.2}$$

Equation (4.2) implies that accuracy can be estimated by counting the instances for which *body* holds and the instances for which both *head* and *body* hold. In practice, the accuracy can be estimated as the relative frequency of covered positive examples over all the examples covered:

$$A(c) = P(e \in E^+|c) = \frac{n^+(c)}{n^+(c) + n^-(c)} \tag{4.3}$$

Where n^+, n^- are the positive and negative examples covered by the rule, respectively.

If we see *head* and *body* as pairs (x, y), we can rewrite (4.2) as:

$$A(body) = \frac{|(x,y) \in E^+ \cap (x,y) \in body|}{|((x,y) \in E^+ \cap (x,y) \in body) \cup ((x,y) \in E^- \cap (x,y) \in body)|} \tag{4.4}$$

where $E^+ = head$. In other words, all the positive examples that we can use come from the facts about the head relation in the knowledge base. Note also, that the *body* is a sequence of literals (other relations that also exist in the knowledge base), which have their arguments chained (bounded) and contain the variables of the *head* x, y. Therefore, we can see the *body* also just as a set of pairs (x, y).

In the case of an incomplete knowledge base, both *head* and *body* might suffer from incompleteness, which means that there are entities which exist in the *head*, but not in the *body* and vice versa. For the entities that exist only in the *body*, the *body* produces pairs, for which we do not know if they are true or false (positive - E^+ or negative examples - E^-). One way to handle this problem is to make the overly pessimistic prediction that all these pairs are corresponding to negative examples, but in this way the rules are being overly penalized. In our approach, while measuring the predictive strength of a rule, we will keep one argument fixed (the input argument) and we will investigate the behavior of the rule only for the pairs in *head* and *body* that share common entities for this argument. Figure 4.1 illustrates the incompleteness in *head* and *body* for the rule $livesIn(x, y)$:-$bornIn(x, y)$; normally the two sets of Figure 4.1 should be completely overlapping with each other.

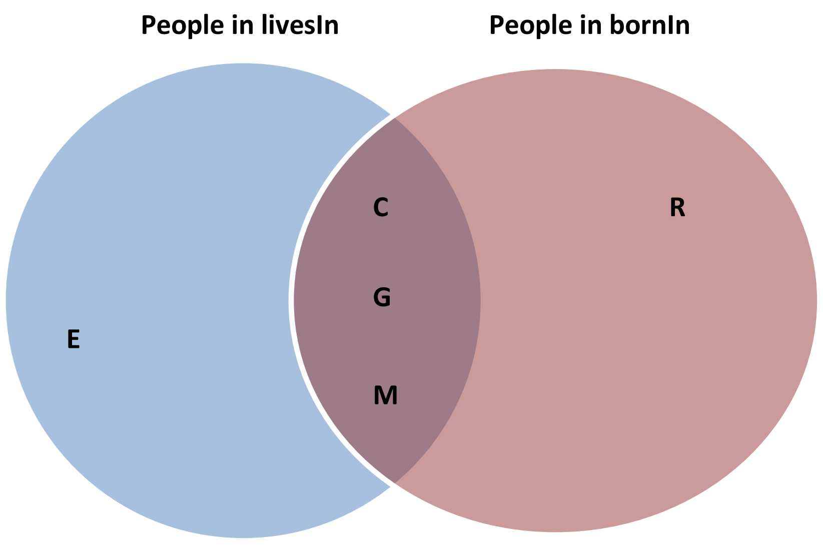


FIGURE 4.1: Sets of persons in the incomplete predicates *bornIn* and *livesIn*

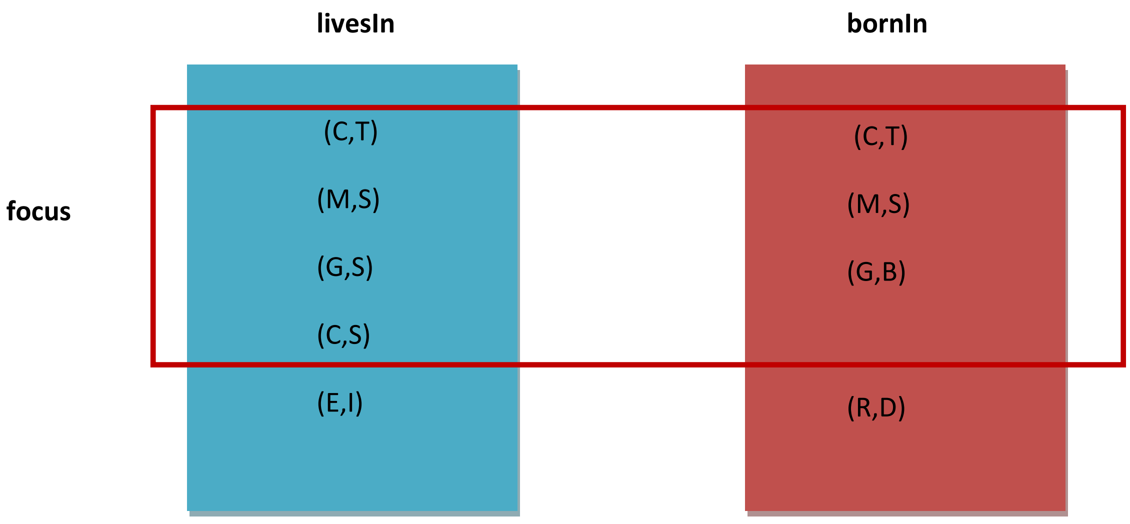


FIGURE 4.2: Facts (and facts of interest) in *head* and *body*.

In Figure 4.2, we can see the exact facts in *head* and *body* of the same rule ($livesIn(x, y)$:-$bornIn(x, y)$) and the subset of pairs that we will investigate for judging the performance of the rule, assuming the first argument as input. Notice that in the following we will concentrate on the facts which share the same 1st argument in *head* and *body* (assuming that the 1st argument is the input argument)

In our approach, we will use as a measure of the "goodness" of the rule (i.e., its predictive strength) the notion of confidence, which is defined as follows:

- Case 1: The body is seen as a mechanism for predicting the (output) variable y, given the (input) variable x, such that the head will be true. Then, by using the definition of conditional probability, the expected accuracy and the confidence will take the following general forms:

$$\begin{aligned} A(c) = P[e \in E^+|c] &= P[(x,y) \in \mathit{head} \mid (x,y) \in \mathit{body} \ \wedge \ \exists(x,z) \in \mathit{head}] \\ &= \frac{P[(x,y) \in \mathit{head} \ \wedge \ (x,y) \in \mathit{body} \ \wedge \ \exists(x,z) \in \mathit{head}]}{P[(x,y) \in \mathit{body} \ \wedge \ \exists(x,z) \in \mathit{head}]} \end{aligned} \tag{4.5}$$

$$\begin{aligned} \mathit{confidence}(\mathit{head}\text{:-}\mathit{body}) &= \frac{|(x,y) \in \mathit{head} \ \cap \ (x,y) \in \mathit{body}, \ \exists(x,z) \in \mathit{head}|}{|(x,y) \in \mathit{body}, \ \exists(x,z) \in \mathit{head}|} \\ &= \frac{|(x,y) \in \mathit{head} \ \cap \ (x,y) \in \mathit{body}|}{|(x,y) \in \mathit{body}, \ \exists(x,z) \in \mathit{head}|} \end{aligned} \tag{4.6}$$

  Note that the only difference between the original definition of accuracy in Equation (4.2) and our definition in Equation (4.5) is the existential part $\exists(x,z) \in \mathit{head}$. We can see that in the nominator of Equation (4.6) this existential part is canceled out by the effect of the join between head and body . In the end, the existential part affects only the denominator by making it smaller and in this way it provides a more optimistic measure for the predictive strength of the rule. In other words, for the example of the rule $\mathit{livesIn}(x,y)$:-$\mathit{bornIn}(x,y)$ in the denominator of confidence we are only considering those persons produced by the *body* (i.e persons for which we know where they are born) which also exist in the *head* (i.e. for which we know also where they live). In this way, we are able to avoid over-penalizing our rules because of incompleteness.

- Case 2: The body is seen as a mechanism for predicting the (output) variable x:

$$\begin{aligned} A(c) = P[e \in E^+|c] &= P[(x,y) \in \mathit{head} \mid (x,y) \in \mathit{body} \ \wedge \ \exists(z,y) \in \mathit{head}] \\ &= \frac{P[(x,y) \in \mathit{head} \ \wedge \ (x,y) \in \mathit{body} \ \wedge \ \exists(z,y) \in \mathit{head}]}{P[(x,y) \in \mathit{body} \ \wedge \ \exists(z,y) \in \mathit{head}]} \end{aligned} \tag{4.7}$$

$$\begin{aligned} confidence(head\text{:-}body) &= \frac{|(x,y) \in head\ \cap\ (x,y) \in body,\ \exists(z,y) \in head|}{|(x,y) \in body,\ \exists(z,y) \in head|} \\ &= \frac{|(x,y) \in head\ \cap\ (x,y) \in body|}{|(x,y) \in body,\ \exists(z,y) \in head|} \end{aligned} \tag{4.8}$$

- Case 3: The body is seen as a mechanism which takes as input both variables and tries to predict if the formula $target(x,y)$ is true:

$$\begin{aligned} A(c) &= P[e \in E^+ | c] \\ &= P[(x,y) \in head \mid (x,y) \in body\ \wedge\ \exists(z,y) \in head\ \wedge\ \exists(x,w) \in head] \\ &= \frac{P[(x,y) \in head\ \wedge\ (x,y) \in body\ \wedge\ \exists(z,y) \in head\ \wedge\ \exists(x,w) \in head]}{P[(x,y) \in body\ \wedge\ \exists(z,y) \in head\ \wedge\ \exists(x,w) \in head]} \end{aligned} \tag{4.9}$$

$$\begin{aligned} & confidence(head\text{:-}body) = \\ & \frac{|(x,y) \in head\ \cap\ (x,y) \in body,\ \exists(z,y) \in head,\ \exists(x,w) \in head|}{|(x,y) \in body,\ \exists(z,y) \in head,\ \exists(x,w) \in head|} = \\ & \frac{|(x,y) \in head\ \cap\ (x,y) \in body|}{|(x,y) \in body,\ \exists(z,y) \in head,\ \exists(x,w) \in head|} \end{aligned} \tag{4.10}$$

If we compare the equations for the three different cases, we will see that the only difference lies in the existential parts.

In general we could define confidence as the ratio of positive training examples covered by the rule c, from now on referred to as *positive examples covered*$(N^+(c))$, over the related examples produced by the rule, from now on referred to as *examples covered* $(N(c))$:

$$confidence = \frac{positive\ examples\ covered}{examples\ covered} = \frac{N^+(c)}{N(c)} \tag{4.11}$$

For the example in Figure 4.2, we have positive examples covered $N^+(c) = 2$, namely, (C,T) and (M,S) out of $N(c) = 3$ examples covered, namely, (C,T), (M,S), (G,B). By applying Equation (4.6), we avoid counting (R,D) in the body of the rule, since we have no evidence if it is true or false. For an illustration see Figure 4.3.

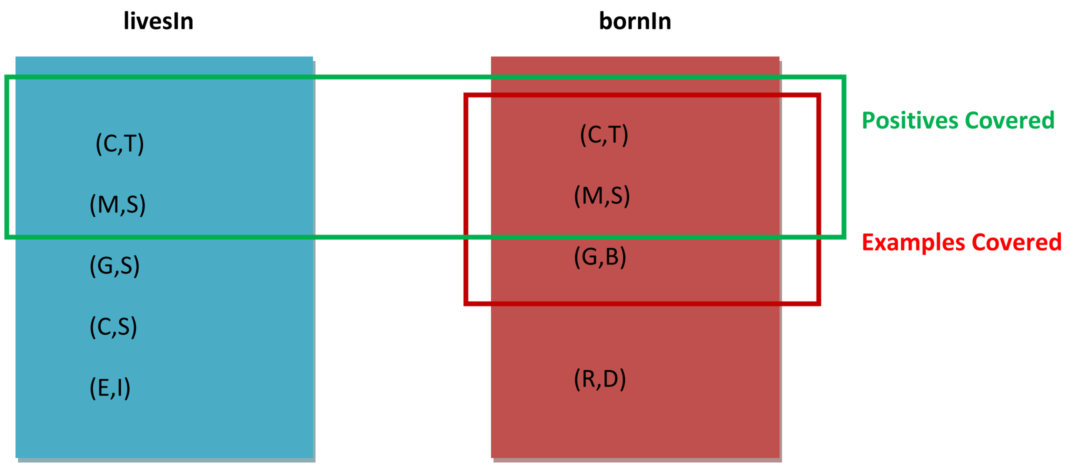


FIGURE 4.3: Illustration of positive examples covered ($N^+(c)$) and examples covered ($N(c)$).

Note that confidence only measures the rule's ability to produce a fact that is correct. For example, the rule is not penalized for not being able to produce all the positive examples that exist in the head (e.g. (C, SB)).

By using the confidence measure, we want to evaluate the rule's ability to predict correctly, given that the rule is able to predict at all and given that there is a way to tell if it predicted correctly.

For counting the positive examples covered ($N^+(c)$) and examples covered ($N(c)$) in our knowledge base, we use the SQL statements of the Figure 4.4, which assumes that the first argument is the input. The positive examples covered ($N^+(c)$) are calculated by a join on both argument of *head* and *body*, whereas the examples covered ($N(c)$) are calculated by counting the facts in the *body* that have in their input argument some entity that also exists in the *head*. In both cases, the group-by statement is used so that we will count distinct facts without an explicit distinct statement (for query optimization reasons). Notice that all the formulas above imply set (and not bag) notation.

Figures 4.5 - 4.7 show SQL statements for calculating examples covered ($N(c)$) for rules with longer bodies, assuming again the first argument as input. Figure 4.5 shows the case of a rule with a new variable introduced, in contrast to Figure 4.6 in which no new variable is introduced. The rule of Figure 4.7 has a new variable which remains unbound (free). Figure 4.8 shows the SQL statement for calculating the examples covered ($N(c)$) with Formula (4.9) for Case 3. Note that if the same variable appears in the body in two different literals, this is immediately translated to a join between these literals by the nested subquery. The "IN" subquery practically expresses

```
SELECT count(*)  examplesCovered
FROM
    (SELECT body.arg1,body.arg2
    FROM facts body
    WHERE body.relation='bornIn'
    AND body.arg1 IN
       (SELECT head.arg1
       FROM facts head
       WHERE head.relation='livesIn')
    GROUP BY body.arg1,body.arg2)
```

```
SELECT count(*)  positivesCovered
FROM
    (SELECT body.arg1,body.arg2
    FROM facts body, facts head
    WHERE body.relation='livesIn'
         AND head.relation='bornIn'
         AND head.arg1=body.arg1
         AND head.arg2=body.arg2
    GROUP BY body.arg1,body.arg2)
```

Figure 4.4: Calculation of positive examples covered ($N^+(c)$) and examples covered ($N(c)$).

the existential part of the Equations (4.6)- (4.9). In other words, it is an equivalent form of a “Where Exists” subquery.

```
SELECT count(*) examplesCovered
FROM
        (SELECT f1.arg1,f2.arg2
         FROM facts f1, facts f2
         WHERE f1.relation='hasChild' AND f2.relation='bornIn'
         AND f1.arg2=f2.arg1
         AND f1.arg1 IN
                      (SELECT f0.arg1
                       FROM facts f0
                       WHERE f0.relation='livesIn')
         GROUP BY f1.arg1,f2.arg2 )
```

Figure 4.5: Calculation of examples covered ($N(c)$) for the rule livesIn(x,y):-hasChild(x,z), bornIn(z,y)

Another main difference of our approach from the standard ILP algorithm (see Section 2.2.7) is that, each time we construct a new clause, we do not remove the positive examples covered ($N^+(c)$) from the training set. In this way, each clause is evaluated on

```
SELECT count(*) examplesCovered
FROM
         (SELECT f1.arg1,f1.arg2
          FROM facts f1, facts f2
          WHERE f1.relation=’diedIn‘ AND f2.relation=’bornIn’
          AND f1.arg1=f2.arg1 AND f1.arg2=f2.arg2
          AND f1.arg1 IN
                      (SELECT f0.arg1
                       FROM facts f0
                       WHERE f0.relation=’livesIn’)
          GROUP BY f1.arg1,f1.arg2 )
```

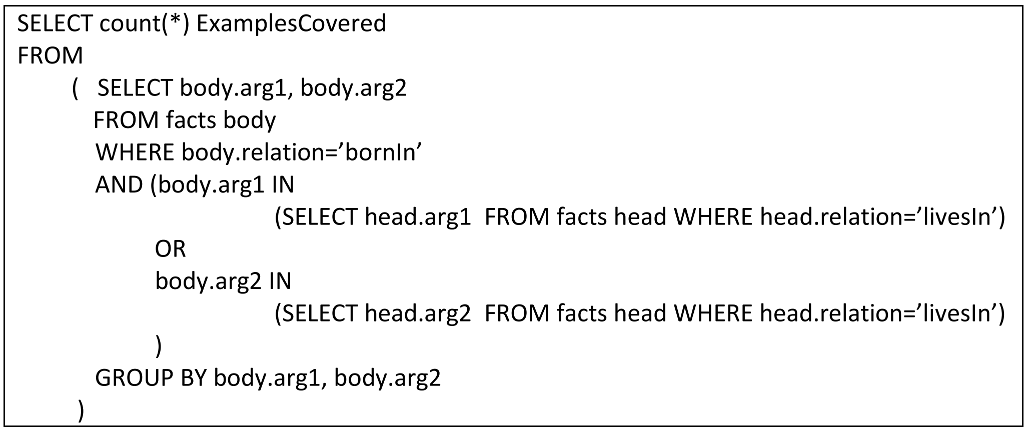
FIGURE 4.6: Calculation of examples covered ($N(c)$) for the rule livesIn(x,y):-diedIn(x,y), bornIn(x,y)

```
SELECT count(*) examplesCovered
FROM
         (SELECT f1.arg1,f1.arg2
          FROM facts f1, facts f2
          WHERE f1.relation=’bornIn’ AND  f2.relation=’hasChild’
          AND f1.arg1=f2.arg1
          AND f1.arg1 IN
                      (SELECT f0.arg1
                       FROM facts f0
                       WHERE f0.relation=’livesIn’)
          GROUP BY f1.arg1,f1.arg2 )
```

FIGURE 4.7: Calculation of examples covered ($N(c)$) for the rule livesIn(x,y):-bornIn(x,y), hasChild(x,z)

the same training data and can be compared with other clauses for its confidence and its support(see Section 4.3).

Another problem that we have to deal with is the absence of negative examples. It is intuitive that when extracting facts from the web, we are only interested in storing true facts. If we combine the absence of explicit negative examples with the incompleteness, it is obvious that we cannot assume that if a fact is not in the knowledge base is definitely false (closed world assumption). Of course, in the case that the head predicate is functional, we are able to infer some negative examples. Consider the rule $diedIn(x, y)$:-$livesIn(x, y)$. *diedIn* is a functional predicate in the sense that each person can only die in one place. Assume that in the *head* we have the fact $diedIn(C, T)$



```
SELECT count(*) ExamplesCovered
FROM
      (  SELECT body.arg1, body.arg2
         FROM facts body
         WHERE body.relation='bornIn'
         AND (body.arg1 IN
                          (SELECT head.arg1  FROM facts head WHERE head.relation='livesIn')
                OR
                body.arg2 IN
                          (SELECT head.arg2  FROM facts head WHERE head.relation='livesIn')
                )
         GROUP BY body.arg1, body.arg2
      )
```

FIGURE 4.8: Calculation of examples covered ($N(c)$) for the rule livesIn(x,y):-bornIn(x,y) for Case 3 Eq. (4.9)

and the *body* produces the fact (C, S). Immediately, the fact $diedIn(C, S)$ can only be false, because in reality C died in T. In other words, the fact $diedIn(C, S)$ is a negative example (an implicit one) which is covered by the rule. This means that in the case of functional head predicates, confidence is calculated in the same way as accuracy in Equation (4.3). For the cases that the head predicate is non-functional, we can still assume functional behavior, while calculating confidence, but the resulting confidence will be a more strict and pessimistic estimate of the true confidence. For a different approach in calculating confidence for non-functional head predicates see Section 4.8.

4.3 Support

Apart from the notion of confidence, which is a measure for the predictive strength of the rule, we will also use the notion of support. We define support as the fraction of instances for which the body is able to make a prediction, and there is a way to tell if it predicted correctly, among the overall positive examples that exist in the head. Nevertheless, we do not care if the rule predicts successfully in the end. Equations (4.12),(4.13),(4.14) are expressing the support analogously for the confidence cases of the Equations (4.6),(4.8),(4.9):

- Case 1: input argument x

$$support(body \rightarrow head) = \frac{|(x, y) \in head, \exists(x, z) \in body|}{|(x, y) \in head|} \tag{4.12}$$

- Case 2: input argument y

$$support(body \rightarrow head) = \frac{|(x,y) \in head, \exists(z,y) \in body|}{|(x,y) \in head|} \tag{4.13}$$

- Case 3: x, y both given as input

$$support(body \rightarrow head) = \frac{|(x,y) \in head, \exists(x,z) \in body, \exists(z,y) \in body|}{|(x,y) \in head|} \tag{4.14}$$

In general, we could define support as the ratio of the possible positive training examples which could be covered by a rule c, from now on referred to as *possible positive examples to be covered* $(E^+(c))$, over the size of the head of the rule (E^+):

$$support = \frac{possible\ positive\ examples\ to\ be\ covered}{facts\ in\ head\ relation} = \frac{E^+(c)}{E^+} \tag{4.15}$$

For the example in Figure 4.2, we have $E^+(c) = 4$ possible positive examples to be covered, namely, (C,T) , (C,S), (G,S), (M,S) out of $E^+ = 5$ facts that exist in the head, namely, (C,T), (C,S), (G,S), (M,S), (E,I). The positive example (E,I) cannot "possibly" be covered by the rule, since the entity E does not exist in the *body*. For an illustration see Figure 4.9. Note that the number of possible positive examples to be covered can be translated as an upper bound for the number of positive examples covered $(N^+(c))$. The positive examples covered $(N^+(c))$ will be equal to the possible positive examples to be covered if the rule predicts correctly the y values for the (common between *head* and *body*) x entities.

In the classical ILP setting, we saw that the algorithm tries to cover all positive examples, which, in our case, are the facts of the head relation. Of course, the number of positive examples covered $(N^+(c))$ can be at most as high as the number of possible positive examples to be covered $(E^+(c))$, which means that, in the end, $E^+(c)$ can be seen as an optimistic upper bound for $N^+(c)$.

Another thing that we should notice is that support grows smaller by adding new literals in the *body*; as we add literals and form more specific clauses, the positive and negative examples covered $(N(c))$ by the clause can only grow smaller, which causes lower values for support. This property (very similar to the anti-monotonicity property of association rule mining [AIS93]) gives us an opportunity for an early pruning of the search space: if a rule has support lower than some threshold, there is no need to check its refinements, because their support will also be below the threshold. This pruning can be applied even while constructing illegal clauses (clauses, in which only one argument of the head appears in the body) as intermediate steps for legal rules, because in the

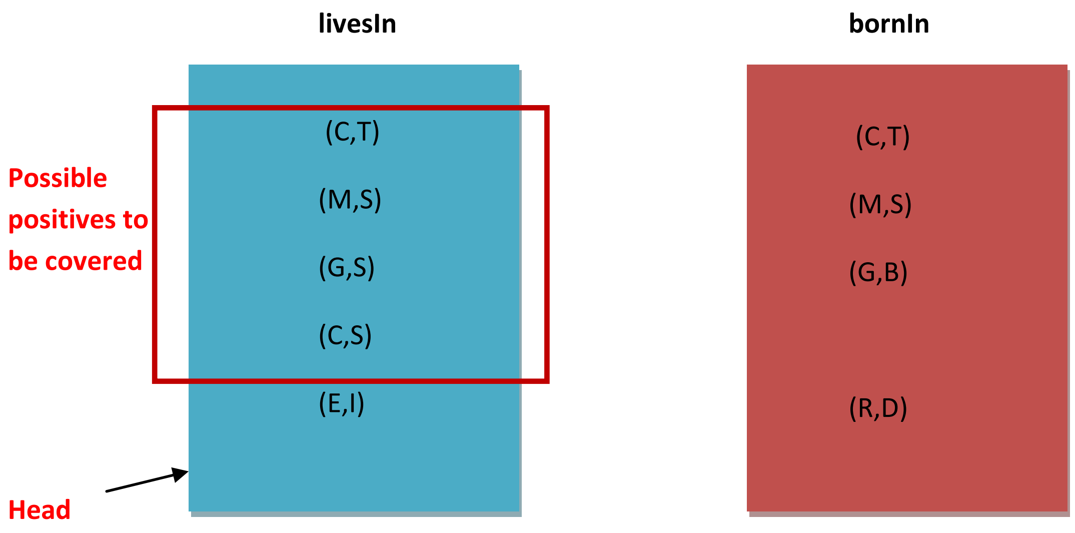


FIGURE 4.9: Illustration of possible positive examples to be covered ($E^+(c)$) and head (E^+).

nominator of support we do not have the positive examples covered ($N^+(c)$), whose calculation requires both arguments of the *head* joined with the *body*, but we have the possible positive examples to be covered ($E^+(c)$), whose calculation requires only a join on the input argument. In Figure 4.10, we can see the SQL statement for calculating $E^+(c)$ of the rule $livesIn(x, y)$:-$bornIn(x, y)$ for Case 1, and in Figure 4.11 for Case 3.

```
SELECT count(*) possiblePositivesToBeCovered
FROM
        (SELECT head.arg1, head.arg2
         FROM facts head
         WHERE head.relation='livesIn'
         AND head.arg1 IN
                 (SELECT body.arg1
                  FROM facts body
                  WHERE body.relation='bornIn') )
```

FIGURE 4.10: Calculation of $E^+(c)$ for the rule livesIn(x,y):-bornIn(x,y)

The intuition behind the confidence measure is how strong the rule is according to the data we have seen already. Support, on the other hand, is a measure of whether the rule will behave in a similar way to unseen data, i.e., to facts about entities that

```
SELECT count(*) possiblePositivesToBeCovered
FROM
        ( SELECT head.arg1, head.arg2
          FROM facts head
         WHERE head.relation='livesIn'
         AND  ( head.arg1 IN
                              (SELECT body.arg1 FROM facts body WHERE body.relation='bornIn')
                  OR
                 head.arg2 IN
                              (SELECT body.arg2 FROM facts body WHERE body.relation='bornIn')
                )
         )
```

FIGURE 4.11: Calculation of $E^{+}(c)$ for Case 3 (Eq. (4.14))

exist only in the *head*, but not in the *body*, as it does to already seen data, i.e., to facts about common entities in *head* and *body*.

4.4 Relationship to Confidence and Support in Association Rule Mining

Association rule mining is a popular and well researched method for discovering interesting relations between variables in large databases. The standard setting includes a set of transactions, each of which includes different items. According to Agrawal [AIS93], the problem can be defined as: "Let $I = I_1, I_2, ..., I_m$ be a set of binary attributes, called items. Let T be a database of transactions. Each transaction t is represented as a binary vector, with $t[k] = 1$ if t bought the item I_k, and $t[k] = 0$ otherwise. There is one tuple in the database for each transaction. Let X be a set of some items in I. We say that a transaction t satisfies X if for all items I_k in X, $t[k] = 1$. By an association rule, we mean an implication of the form $X \rightarrow I_j$, where X is a set of some items in I, and I_j is a single item in I that is not present in X. The rule form $X \rightarrow I_j$ is satisfied in the set of transactions T with the confidence factor $0 \leq c \leq 1$ iff at least $c\%$ of transactions in T that satisfy X also satisfy I_j. "

According again to Agrawal's paper [AIS93]: "The support for a rule is defined to be the fraction of transactions in T that satisfy the union of items in the consequent and antecedent of the rule. Support should not be confused with confidence.

While confidence is a measure of the rule's strength, support corresponds to statistical significance."

The set of possible item-sets is the power set over I and has size $2^n - 1$ (excluding the empty set which is not a valid item-set). Although the size of the powerset grows exponentially in the number of items n in I, efficient search is possible using the downward-closure property of support (also called anti-monotonicity) which guarantees that for a frequent item-set also all its subsets are frequent, and thus for an infrequent item-set, all its supersets must be infrequent.

The notion of support, as we have defined it for our setting, maintains the anti-monotonicity property in the following sense: the body of the rule can be seen as an item-set and if it is "infrequent" (with low support), then all its super-sets will also be infrequent (their support cannot be higher).

Luc de Raedt and Hannu Toivonen have investigated the unification of ILP and association rule mining. In [DT98] they argue that association rule and frequent set discovery can be seen as well-controlled subtasks of the full ILP framework, demonstrate how ILP can solve both existing and new data mining problems and analyze the gradual change in the trade-off between expressivity and efficiency, as one moves from a frequent set discovery problem to ILP.

4.5 Speciality Ratio

The third measure which we are going to use is the *Speciality Ratio*. The need for the introduction of this measure has to do with the absence of explicit negative examples and the danger of producing overly general rules, which still have high confidence and high support. For example, the rule $\mathit{politicianOf}(x,y)$:-$\mathit{bornIn}(x,z), \mathit{locatedIn}(z,y)$ is overly general, because it produces politicians for all people born somewhere and therefore is not a rule of interest. At the same time, this rule is expected to exhibit a high support, since we expect a large percentage of the persons in the $\mathit{politicianOf}$ relation to be also contained in the bornIn relation (the possible positive examples to be covered ($N^+(c)$) is expected to be high) and it can also exhibit a high confidence, since, while counting the examples covered ($N(c)$), we do not take into consideration the entities that do exist in the *body*, but not in the *head*, i.e., the people for which we know were they are born, but they are not politicians. In order to avoid accepting such rules, we will use the measure of speciality ratio, which we define as follows:

$$\mathit{Speciality\ Ratio} = \frac{\mathit{examples\ covered}}{\mathit{Size\ of\ body}} = \frac{N(c)}{B(c)} \tag{4.16}$$

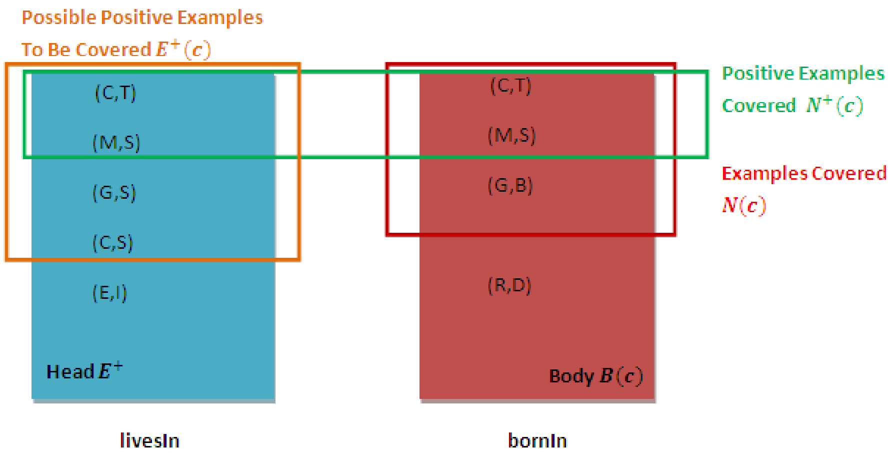


FIGURE 4.12: Illustration of nominators and denominators for all measures.

In general, we can see the way in which we are calculating confidence as: first sampling the *body* and getting only the facts in the examples covered ($N(c)$) as sample and then checking how well the rule behaves in this sample. Of course, this means that we need to be sure that the sample is large enough in comparison to the original body.

In our case, if the speciality ratio grows very low, this means that the rule produces many pairs for which we do not know anything, i.e., too many entities of the input argument are not appearing in the *head* at all. This means either that the *head* is very incomplete or that the rule is very general, like in our example. So, if speciality ratio gets lower than some threshold, we should mark the rule as potentially overly general and we will try to make the rule more special by expanding it. So, for the previous example-rule $politicianOf(x, y)$:-$bornIn(x, z), locatedIn(z, y)$, by refining it, we can get a more interesting rule like $politicianOf(x, y)$:-$bornIn(x, z), locatedIn(z, y), type(x, Politician)$.

In Figure 4.12 we can see the illustration of all nominators and denominators for all the measures, which we have introduced so far, namely, confidence, support and speciality ratio.

4.6 Pruning Opportunities

As we have already seen, while refining a given clause, the overall examples covered ($N(c)$) by the refinement cannot be more than those of the initial clause. Figure 4.13 illustrates how the coverage of a rule changes during the refinement procedure. The rectangular with label 1 corresponds to the rule $livesIn(x, y)$:-$bornIn(x, y)$, the one with label 2 to the rule $livesIn(x, y)$:-$bornIn(x, y), diedIn(x, y)$ and the one with label 3 to the rule $livesIn(x, y)$:-$bornIn(x, y), diedIn(x, y), politicianOf(x, y)$.

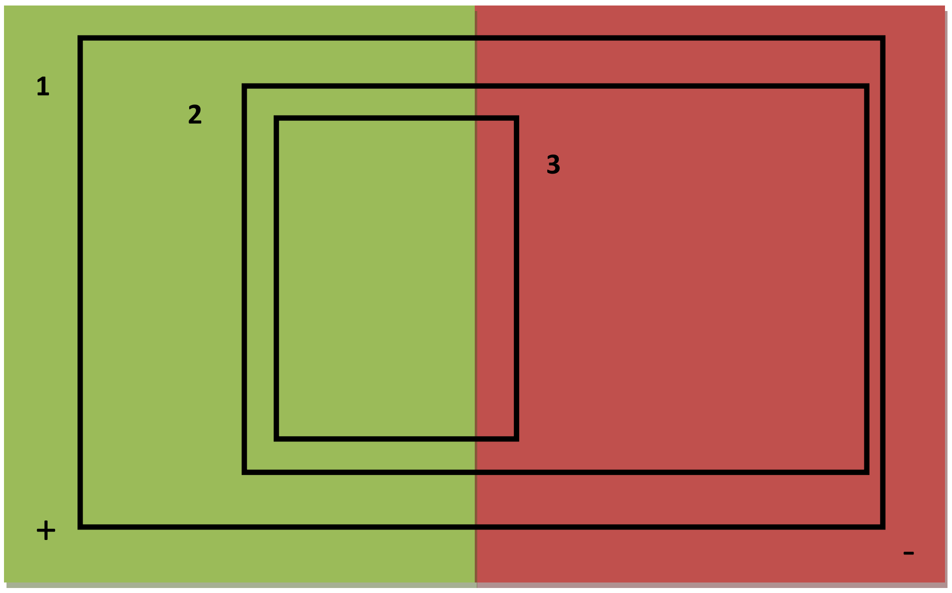


Figure 4.13: The change in coverage of positive (green) and negative (red) examples while refining a rule.

As it is obvious from Figure 4.13, the ratio of positive and negative examples covered by the rule does not exhibit any monotonicity property. This means that pruning by confidence is not safe. On the other hand, the overall number of examples covered ($N(c)$) can only become smaller (or at most remain the same). This means that we can either stop considering some clause, if its possible positive examples to be covered ($E^+(c)$) are below some absolute value. E.g., we want to test the strength of a rule for at least 100 facts. Or if its support is below some ratio. E.g., we want to test the strength of the rule for at least the 10 percent of the facts which exist in the *head*. In the end, we output a rule as a result of our learning algorithm, if it is able to pass:

1. the support or possible positive examples to be covered ($E^+(c)$) threshold
2. the confidence and positive examples covered ($N^+(c)$) threshold
3. the speciality ratio threshold

The learning algorithm will be described in details in Section 5.4.5.

4.7 The Choice of the Input Argument

So far we have seen that most of the measures we have introduced depend on the choice of the input argument. But how can we decide which argument to use as input variable?

We argue that it makes sense to take into consideration the way the facts were extracted. For the following, we assume that our knowledge base consists of facts extracted from web pages like Wikipedia, just like the YAGO knowledge base [SKW07]. Assume that we want to learn the relation $bornIn(person, city)$. The body of the rule produces some pairs (x, y), and we need to decide which should be the input variable, i.e., we need to decide the presence of which variable cause us to "demand" correct prediction from the rule. In our example, facts about the head predicate ($bornIn$) are more probable to be extracted from Wikipedia pages for some person, which means that if there is a fact about a person x, i.e., the Wikipedia page for this person is visited, even if the y that we extracted from the page does not appear in the *body* at all, it is not an excuse for the rule to predict incorrectly. In other words, we make the assumption that our knowledge base is incomplete, because the extractor was not able to visit the Wikipedia pages of some persons, or it was not able to extract the related relations of the page. I.e, we do not count as negative examples, facts in the body for which the x variable does not appear in the head at all. But if the web page was visited and the extractor managed to extract a fact for the specific relation, the incompleteness is no more an excuse for the rule not predicting correctly. Hence, we argue that if the rule cannot predict correctly, it should be penalized. Figure 4.1 shows how support and confidence are changing according to the choice of the input argument for some example rules.

The problem is that in some cases it is not very clear, if the extraction took place in the Wiki page of the first argument or of the second. For example, consider the relation $isMarriedTo$. Assume that the body creates the pair("Tom Cruise", "Nikole Kidman") and that we take the first argument as input. The extractor has visited Tom Cruise's web page, but was only able extract that he is married to Katie Holmes. In Nikole Kidman's web page there is the info that she was married to Tom Cruise, but, unfortunately, the extractor never visited her web page. In such cases, we can still calculate confidence and support by using Equations (4.9) and (4.14), respectively.

In general, we will leave the choice for specifying the input argument to the knowledge engineer, i.e., the user. Nevertheless, in cases that the head predicate exhibits functional properties (i.e., we observe only one fact for each entity of the argument under consideration), we are able to suggest as input argument the argument which exhibits this functional behavior.

Input Argument	**Rules**
1st Argument (x)	$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.9822 support: 0.3799 $familyNameOf$: no rule learned $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.9001 support: 0.4593
2nd Argument (y)	$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.8364 support: 0.4459 $familyNameOf(x,y)$:-$hasChild(y,z), familyNameOf(x,z), (z \neq y)$ confidence: 0.6346 support: 0.0025 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.9529 support: 0.4586
Both Arguments (x,y)	$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.2838 support: 0.9199 $familyNameOf(x,y)$:-$hasChild(y,z), familyNameOf(x,z), (z \neq y)$ confidence: 0.4423 support: 0.0054 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.8151 support: 0.5142

Table 4.1: The effect of input argument in support and confidence.

4.8 Support and Confidence: Taking Incompleteness into Consideration

As we have already seen, it is possible that both the *head* and the *body* of the rule suffer from incompleteness. In the way we have defined support and confidence, we ignore entities of the input argument that exist in the *body*, but not in the *head* and vice versa, but our measures are still affected by incompleteness in the following way: we might have some facts about some entity of the input argument, but we are still missing some facts for this entity. For example, the fact $hasChild(P, C)$ exists in the *head* of the rule, but we are missing the second parent of C.

In other words, for each entity of the input argument, we might observe in our data on average a specific number of facts of a given predicate (we will call this measure multiplicity $mult$), but in the real world each entity might be connected with a higher number of facts on average (we will call this measure ideal multiplicity $idealMult$). Or, in other words, for each entity and a specific predicate, we have $mult(head)$ many facts in the database, and we are missing $idealMult(head) - mult(head)$ many facts. Similarly, the *body* of a rule produces $mult(body)$ many facts which we can observe, but also $idealMult(body) - mult(body)$ many facts that we do not know of.

In the special case where $idealMult(head) = 1$, every entity of the input argument can only be associated with one fact of a given predicate, and in practice this means that this predicate is functional for the specific input argument. In this case, as we have already seen, if the *body* generates for the specific entity something different than the fact that we have in the *head*, we are sure that the produced fact is false, i.e., the rule covers a negative example. In all other cases, where the head predicate is non functional, we cannot really be sure whether the produced fact is true or false. This means that the confidence measure, as we have defined it in Equation (4.11), might be over-penalizing towards the rules, assuming as negative examples those produced by the *body* which do not exist in the *head*.

On the other hand, this kind of incompleteness exists also in the *body*; the *body* might produce in reality more facts , than those that we are able to observe, many of which might be false. Again, this means that our confidence definition might be too optimistic for some rules.

In the following, we are making the assumption that the average observed multiplicity and the average ideal multiplicity for both *head* and *body* are given. We also assume that we have fixed our input argument.

We know that, for each entity of the *head*, we know $mult(head)$ facts. This means that on average we have $facts\ in\ head/mult(head)$ distinct entities in the *head*. For each of these entities, we are missing $idealMult(head) - mult(head)$ facts on average, which are all positive examples. This means that the *real head size* (E^+) will be:

$$E^+ = \frac{idealMult(head)}{mult(head)} \cdot E_k^+$$

where E_k^+ is the *known or observed head size*, namely the number of facts that we have in our knowledge base. If we define as *multiplicity ratio (multRatio)* of the *head*:

$$multRatio(head) = \frac{idealMult(head)}{mult(head)}$$

then the formula for the real head size becomes:

$$E^+ = multRatio(head) \cdot E_k^+$$

If we assume that the possible positive examples to be covered by a rule c exhibit the same properties like the *head* (in terms of $mult$ and $idealMult$), then the real possible positive examples to be covered ($E^+(c)$) are:

$$E^+(c) = \frac{idealMult(head)}{mult(head)} \cdot E_k^+(c) = multRatio(head) \cdot E_k^+(c)$$

where $E_k^+(c)$ are the *known or observed possible positive examples to be covered*, namely what we were able to count in our knowledge base. Then, support becomes:

$$support = \frac{E^+(c)}{E^+} = \frac{multRatio(head) \cdot E_k^+(c)}{multRatio(head) \cdot E_k^+} = \frac{E_k^+(c)}{E_k^+}$$

As we see, support remains the same as before. Similarly, the speciality ratio will also remain the same.

So far, we have seen that we are able to count $E_k^+(c)$ possible positive examples to be covered, but in reality we should have had $E^+(c)$. In other words, we are *missing* the following possible positive examples to be covered ($E_m^+(c)$):

$$E_m^+(c) = E^+(c) - E_k^+(c)$$

From the known possible positive examples to be covered ($E_k^+(c)$), the rule covers *known positive examples covered* ($N_k^+(c)$), which is the nominator of the initial confidence formula. From the missing possible positive examples to be covered ($E_m^+(c)$) we do not know how many are covered by the rule. Perhaps all of them, perhaps none of them. Here we will make the assumption that the rule behaves similarly for the $E_k^+(c)$ and for the $E_m^+(c)$, which means that the ratio

$$\frac{\textit{positive examples covered}}{\textit{possible positive examples to be covered}}$$

should remain the same for the known and the missing case. This means that the *real positive examples covered* ($N^+(c)$) should be:

$$N^+(c) = N_k^+(c) + \frac{N_k^+(c)}{E_k^+(c)} \cdot E_m^+(c)$$

On the other hand, we have seen that incompleteness also exists in the body. This means that for each entity in the body, we can observe $mult(body)$ many facts, and we miss $idealMult(body) - mult(body)$ facts. If we assume that the examples covered ($N(c)$) exhibit the same properties like the body then:

$$N(c) = \frac{idealMult(body)}{mult(body)} \cdot N_k(c) = multRatio(body) \cdot N_k(c)$$

where $N_k(c)$ are the known or observed examples covered by the rule c. Then, of course, the missing examples covered by the rule c will be:

$$N_m(c) = N(c) - N_k(c)$$

Finally, confidence takes the form:

$$confidence = \frac{multRatio(head) \cdot N_k^+(c)}{multRatio(body) \cdot N_k(c)} \tag{4.17}$$

Figure 4.14 is an illustration of the above mentioned sets. The sets which contain true facts are noted with (+) and those which contain false facts with (-).

In the previous discussion, we made the assumption that the average observed and ideal multiplicity are known for both the *head* and *body*. Although, the observed average multiplicity can be easily calculated from the data, the ideal average multiplicity

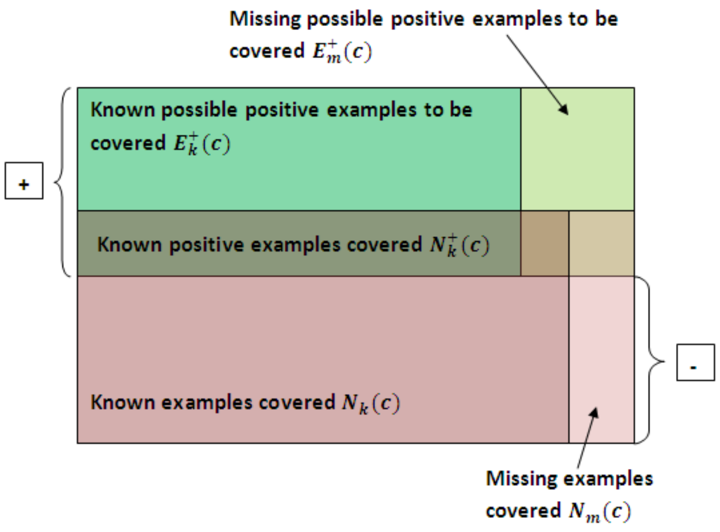

Missing possible positive examples to be covered $E_m^+(c)$
Known possible positive examples to be covered $E_k^+(c)$
+
Known positive examples covered $N_k^+(c)$
Known examples covered $N_k(c)$
-
Missing examples covered $N_m(c)$

FIGURE 4.14: An illustration of the known and missing facts for the calculation of confidence with the Equation (4.17).

is generally not known, which means that we need to estimate it in some way. Figure 4.15 shows the distribution for the *hasChild* predicate both if we assume the first and the second argument as input argument. Notice the incompleteness in the *hasChild* relation: for the case, in which we assume the second argument as input, practically we expect every person to have a multiplicity of 2, because each person has two parents, and so we would expect two $hasChild(x, y)$ facts for each person y. Nevertheless, this is not observed from the data. Table 4.2 shows the average observed multiplicities for both arguments of some relations in YAGO [SKW07]. Figures 4.16 - 4.19 show the multiplicity distributions for various relations. For the ideal average multiplicity, we will make the assumption that it is the average multiplicity observed from the data plus a quantity, which is again unknown, and we will assume that it is equal to one standard deviation σ of the observed distribution:

$$idealMult = mult + \sigma \tag{4.18}$$

The results of the above approach can vary, according to how misleading the observed data and their distribution are. In Figure 4.3, we can see some selected rules and their confidences computed with Formulas (4.17) and (4.11), respectively. As we can see the rule for the *politicianOf* relation gets a desirable, lower confidence with the improved formula. On the other hand, the rules for the *isMarriedTo* and *hasPredecessor* relations are practically properties of these relations (*isMarriedTo* is symmetric and *hasPredecessor* is the inverse relation of *hasSuccessor*) and so they should have a confidence near 1.0. Nevertheless, this is not the case if we take as input the first argument.

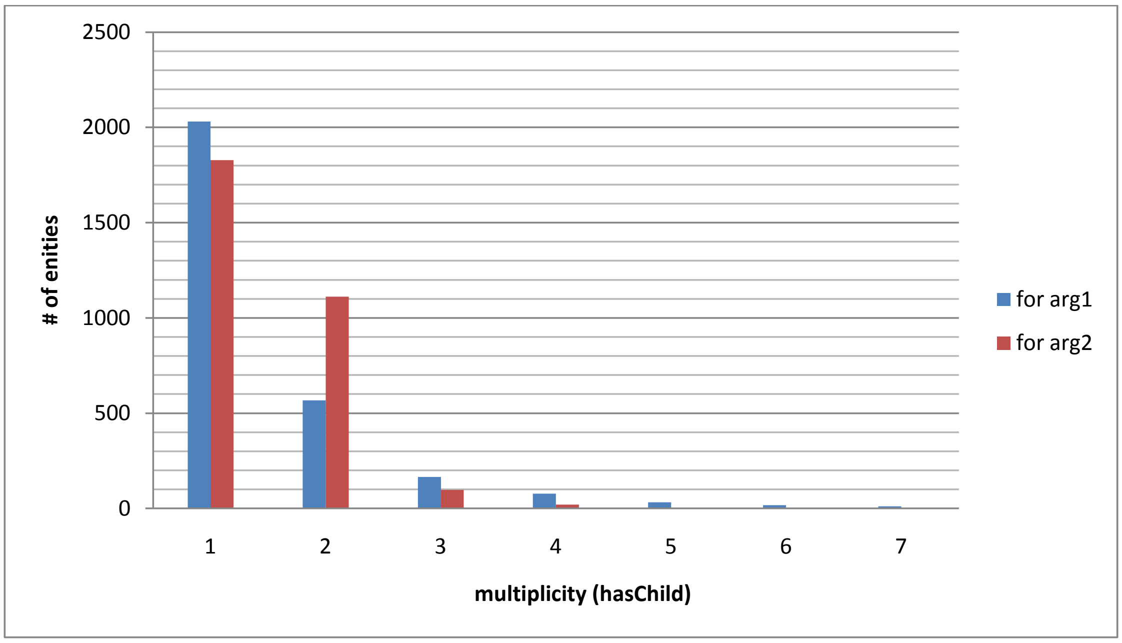


Figure 4.15: The multiplicity distributions for the hasChild relation.

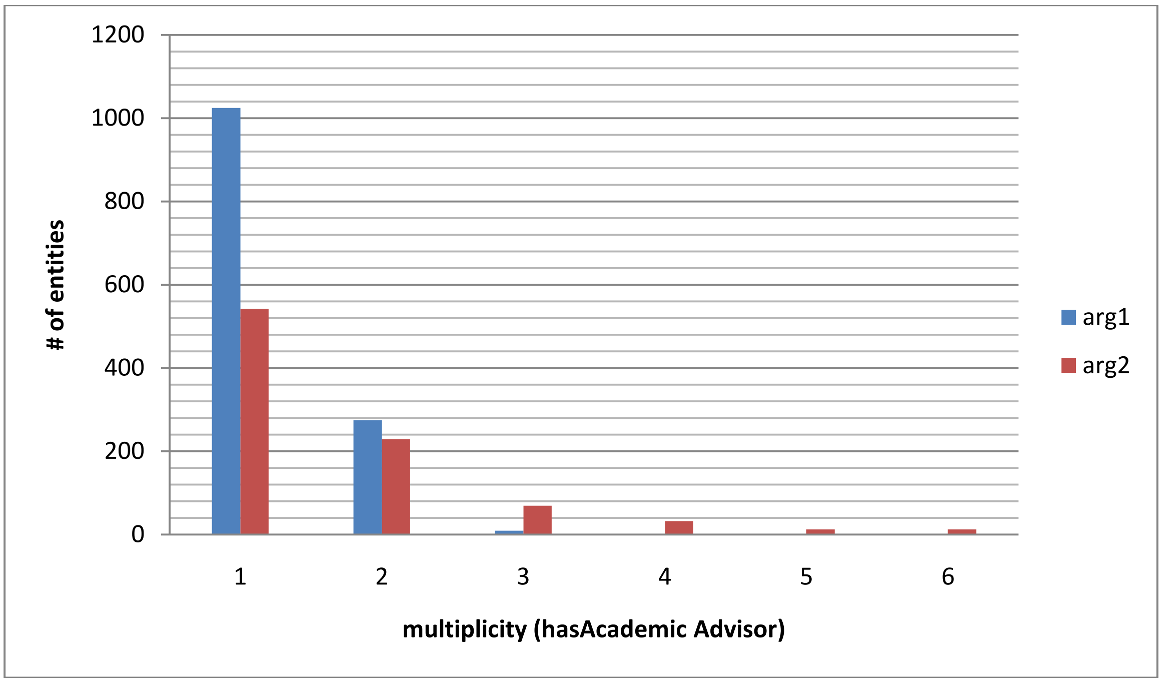


Figure 4.16: The multiplicity distributions for the hasAcademicAdvisor relation.

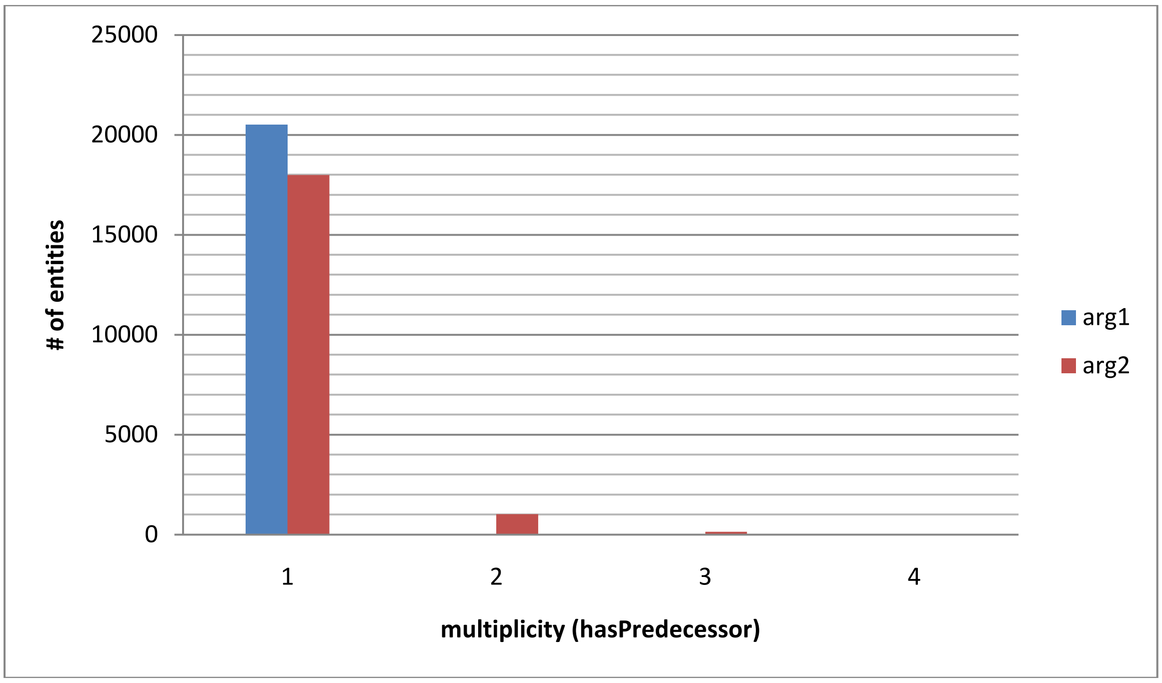


FIGURE 4.17: The multiplicity distributions for the hasPredecessor relation.

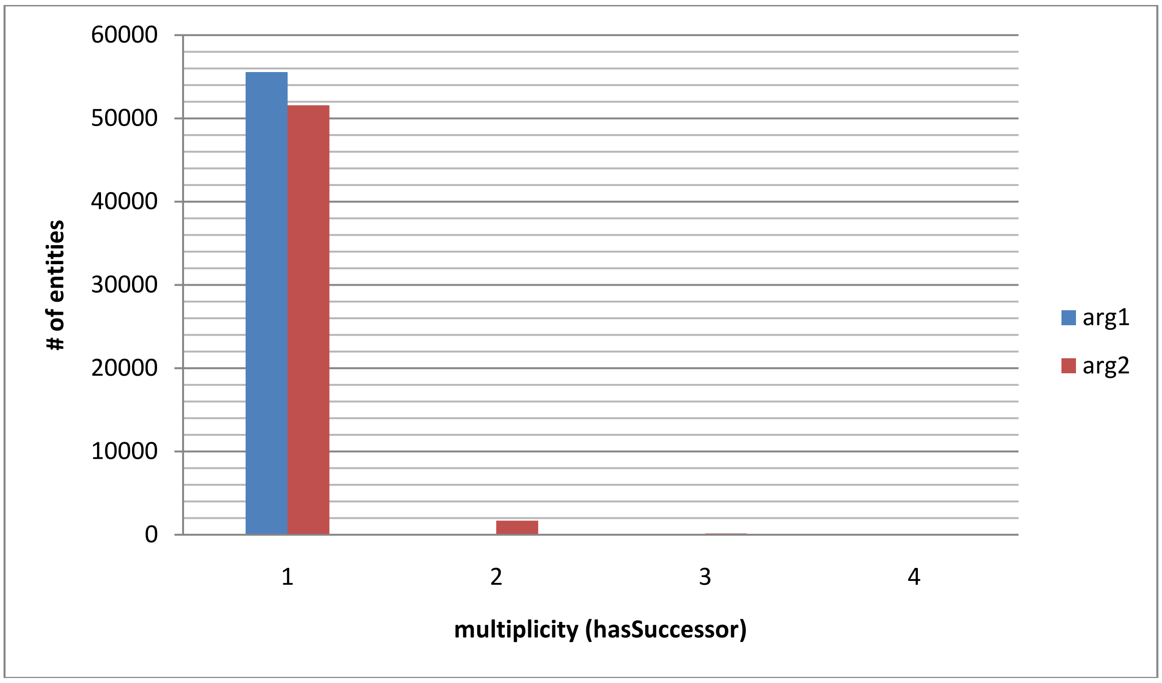


FIGURE 4.18: The multiplicity distributions for the hasSuccessor relation.

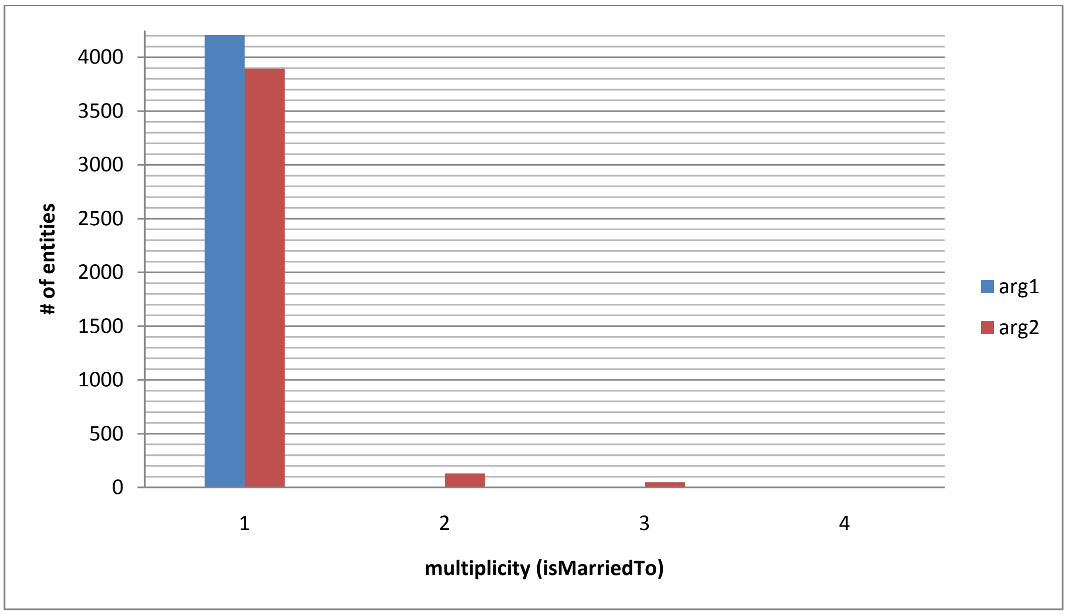


FIGURE 4.19: The multiplicity distributions for the isMarriedTo relation.

Relation	**# of Facts**	**multiplicity for 1st arg**	**multiplicity for 2nd arg**
actedIn	28836	2.046	1.1129
bornIn	36188	1.0	2.9779
bornOnDate	441273	1	10.1901
diedIn	13618	1	3.0907
diedOnDate	205469	1	8.1710
familyNameOf	569410	3.014	1
givenNameOf	568852	9.795	1
graduatedFrom	4967	1	4.6420
hasAcademicAdvisor	1599	1.223	1.7552
hasCapital	1368	1	1.1324
hasChild	4453	1.527	1.4542
hasPredecessor	20514	1	1.0709
hasSuccessor	55534	1	1.0396
isCitizenOf	4865	1	31.1858
isLeaderOf	2886	1.813	1.6472
isMarriedTo	4207	1	1.0415
livesIn	14710	1.000203	3.0942
locatedIn	60261	1.00107	21.2936
worksAt	1401	1	2.6634

TABLE 4.2: Average observed multiplicities

Input Argument: 1	$politicianOf(x, y)$:-$hasSuccessor(x, z), politicianOf(z, y), (z \neq x)$ "Improved" Confidence: 0.85 Original Confidence: 0.97 $hasPredecessor(x, y)$:-$(x \neq y), hasSuccessor(y, x)$ "Improved" Confidence: 0.71 Original Confidence: 0.87 $isMarriedTo(x, y)$:-$(x \neq y), isMarriedTo(y, x)$ "Improved" Confidence: 0.73 Original Confidence: 0.90
Input Argument: 2	$isMarriedTo(x, y)$:-$(x \neq y), isMarriedTo(y, x)$ "Improved" Confidence: 0.99 Original Confidence: 0.95 $hasPredecessor(x, y)$:-$(x \neq y), hasSuccessor(y, x)$ "Improved" Confidence: 0.99 Original Confidence: 0.94

Table 4.3: Confidence values calculated by the equations (4.17)(Improved Confidence) and (4.11) (Original Confidence)

This happens because we are severely misled by the observed distributions for the multiplicity of the first argument. Figures 4.19 and 4.17 imply a functional behavior for the first argument of the $isMarriedTo$ and $hasPredecessor$ relations, i.e., their average observed multiplicity equals 1 which also causes the average ideal multiplicity to be estimated as 1. This means that both rules cannot benefit from the incompleteness of the rule, because according to the data observed there is no incompleteness for the head, and both are penalized for the incompleteness of the body. On the other hand, if we assume the second argument as input, the situation is reversed and both rules get a confidence near 1.0, which is more intuitive for such rules.

Chapter 5

Algorithmic Framework

In this chapter we will discuss the main algorithmic framework of our rule learner. As we have already seen in Section 2.2.8, the main two sources of complexity for ILP are the branching factor and the subsumption test. In Sections 5.1- 5.3, we present preprocessing steps, whose aim is to cope with these complexity sources. In Sections 5.1 and 5.3, we discuss our decisions that lead to a relatively low branching factor, whereas in Section 5.2 we present a sampling method in order to perform a low number of subsumption checks. In Section 5.4 we discuss issues related to the core of our algorithm.

5.1 Relation Preprocessing

As we have already seen in Section 4.3, *support* exhibits a behavior similar to the anti-monotonicity property of support in association rule mining. That is, as more literals are added to the body of the rule during the specialization procedure, the support can only grow smaller or, in the best case, remain the same. In addition to that, we have seen that the possible positive examples to be covered ($E^+(c)$) are an upper bound for the positive examples covered ($N^+(c)$), and so the requirement for a high support (or at least for a high number of possible positive examples to be covered ($E^+(c)$)) is reasonable. These two remarks motivate us for early pruning of branches in the search space. For example, if the intermediate rule

$$livesIn(x, y)\text{:-}hasChild(x, z)$$

cannot pass the support threshold, or if the possible positive examples to be covered ($E^+(c)$) is lower than some absolute value, then any extension of this rule like

$$livesIn(x,y) \text{:-} hasChild(x,z), bornIn(z,y)$$
$$livesIn(x,y) \text{:-} hasChild(x,z), bornIn(z,y), diedIn(z,y)$$
$$livesIn(x,y) \text{:-} hasChild(x,z), livesIn(z,y)$$

cannot be better with respect to support. Note that, although the initial rule is not a valid rule, in the sense that the variable y that appears in the head predicate does not exist in the body, we are still able to calculate support and prune the branch early.

From the above example, it is obvious that it could be very beneficial for the efficiency of the algorithm to have a preprocessing step, in which we can calculate the possible positive examples to be covered ($E^+(c)$), or the support for pairs of relations joined on a specific argument, and then reuse this information in order to avoid exploring some branches in the search space. For example, if we had computed in a preprocessing step the overlap of entities on the first argument of $livesIn$ and on the first argument of $hasChild$, and we had seen that there is not enough overlap, then during the main ILP algorithm we could have avoided investigating the entire set of branches under the rule:

$$livesIn(x,y) \text{:-} hasChild(x,z)$$

In this way, we can keep for each relation and each argument of this relation, a list of relations and their specific arguments that make sense (with respect to support) to be used in a join together. Algorithms 5.1 and 5.2 demonstrate the main idea of the preprocessing step. The method *checkEntityOverlap* of Algorithm 5.1 checks if two given relations can pass the threshold for support and for the number of facts in $E^+(c)$ for specific arguments.

5.1.1 Exploiting Types

In Algorithm 5.2, we are performing an exhaustive check between all pairs of relations and possible bindings between their arguments. Nevertheless, we are able to take advantage of the information about the types of the arguments, in case this information is provided. For example, the join between an argument of type *person* and an argument of type *location* will be empty. So, for example, we can skip the check for the overlap between the first argument of the relation $livesIn(person, location)$ and the second argument of the relation $bornIn(person, location)$. Even for knowledge bases which include a type hierarchy that is not entirely flat, like YAGO [SKW07],

Algorithm 5.1: The method checkEntityOverlap.

Data: r_i, r_j: the relations
arg_i, arg_j; the corresponding arguments
$suppThres$: the threshold for support
$EThres$: the threshold for $E^+(c)$

Result: **true** if the relations pass the test, **false** otherwise

```
1 $Overlap := |r_i.arg_i \cap r_j.arg_j|$
2 if $\frac{Overlap}{min(|r_i|,|r_j|)} > suppThres$ or $Overlap > EThres$ then
3     return true
4 else
5     return false
```

Algorithm 5.2: The Preprocessing Algorithm

Data: r: a set of relations
$suppThres$: the threshold for support
$EThres$: the threshold for $E^+(c)$

Result: $L_{m,n}(i)$ (where $m, n = \{1, 2\}$ are the arguments) the lists of joinable relations for each relation r_i

```
1  foreach $r_i$ in $r$ do
2      foreach $r_j$ in $r$ and $j \geq i$ do
3          if $r_i = r_j$ then
4              sameFlag:=true
5          else
6              sameFlag:=false
7          if $sameFlag$ then
8              $L_{1,1}(i) \leftarrow L_{1,1}(i) \cup r_i$
9              $L_{2,2}(i) \leftarrow L_{2,2}(i) \cup r_i$
10         else
11             if checkEntityOverlap($r_i, r_j, arg1, arg1, suppThres, EThres$) then
12                 $L_{1,1}(i) \leftarrow L_{1,1}(i) \cup r_j$
13                 $L_{1,1}(j) \leftarrow L_{1,1}(j) \cup r_i$
14             if checkEntityOverlap($r_i, r_j, arg2, arg2, suppThres, EThres$) then
15                 $L_{2,2}(i) \leftarrow L_{2,2}(i) \cup r_j$
16                 $L_{2,2}(j) \leftarrow L_{2,2}(j) \cup r_i$
17         if checkEntityOverlap($r_i, r_j, arg1, arg2, suppThres, EThres$) then
18             $L_{1,2}(i) \leftarrow L_{1,1}(i) \cup r_j$
19             $L_{2,1}(j) \leftarrow L_{1,1}(j) \cup r_i$
20         if checkEntityOverlap($r_i, r_j, arg2, arg1, suppThres, EThres$) then
21             $L_{2,1}(i) \leftarrow L_{2,2}(i) \cup r_j$
22             $L_{1,2}(j) \leftarrow L_{2,2}(j) \cup r_i$
```

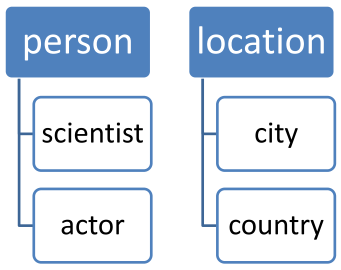


FIGURE 5.1: An example of a simple type hierarchy

we can still take advantage of the types by checking the overlap between arguments that have a common ancestor in the type hierarchy, and by avoiding tests between the arguments of types that do not share any common ancestor. Figure 5.1 shows a simple example of a type hierarchy. In our implementation, we will try to join arguments of types *scientist* and *actor*, because they have the common ancestor *person*, or *scientist* and *person*, because again the *person* type is a common ancestor. For example, we should investigate the overlap for the first arguments of the relations $hasAcademicAdvisor(scientist, scientist)$ and $worksAt(person, institution)$, or of the relations $hasAcademicAdvisor(scientist, scientist)$ and $actedIn(actor, movie)$, since there might be people that were both scientists and actors (types which are siblings in the hierarchy might still have high overlap). On the other hand, we can skip the tests for arguments of type *actor* and *city*, for example. In the case that the type hierarchy has only one root, this approach cannot work. Imagine, for example, that in Figure 5.1 both *person* and *location* were sub-types of the super-type *entity*. In this case, again we need to check all argument pairs of all types, unless the knowledge engineer informs the system that it should not take into consideration the type "entity" when checking for common ancestors in the type hierarchy.

Existing ILP systems like Aleph and Progol [MBR95] are also taking advantage of information about types in order to reduce the search space. For example, in Aleph the user is requested to include in the background knowledge file the following:

- **Type Specifications**. Types have to be specified for every argument of all predicates to be used in constructing a hypothesis. For Aleph, types are just names, and no type-checking is done. Variables of different types are treated differently, even if one is a sub-type of the other, which can be problematic for non-flat type hierarchies.
- **Determination statements**. The user needs to declare the predicates that can be used to construct a hypothesis. In other words, the user should know beforehand which predicates could be used in the body of a clause with a specific head

predicate. Apart from that, for every background knowledge file in Aleph (and for every run) it is allowed to use a determination statement for only one head predicate. So, if the user wants to learn clauses for many different head predicates, he needs to prepare separate files with separate determination statements for each head predicate. As it is obvious, Aleph is trying to restrict the search space by restricting the candidate relations that can possibly participate in the body of the clause. So, for example if we are interested in learning the target relation $livesIn(x, y)$, the determination statement might include the relations $hasChild$, $bornIn$, $diedIn$, $politicianOf$, because the user assumes that they might be relevant, or because he is sure in some way that they can form some indeed good clauses like the following:

$$\begin{gathered} livesIn(x, y)\text{:-}bornIn(x, y) \\ livesIn(x, y)\text{:-}diedIn(x, y) \end{gathered} \tag{5.1}$$

In our setting, the afore-mentioned rules will be constructed if there is a large overlap in the entities of the arguments of the pairs $livesIn-bornIn$ and $livesIn-diedIn$, but if the overlap in the entities of the arguments of the pair $bornIn-diedIn$ is low, we will avoid constructing the rule:

$$livesIn(x, y)\text{:-}bornIn(x, y), diedIn(x, y)$$

(Unlike Aleph, which will have to investigate also this path).

- **Mode declarations**.The user needs to define the number of successful calls to every predicate and to specify, for every argument of each predicate, if this is an input or an output argument. For example, the declaration

  $$\text{:-}mode(1, mult(+integer, +integer, -integer))$$

  implies that the multiplication predicate $mult$ can be once successfully called, and that the first two arguments are input arguments of type integer, whereas the third argument is an output argument. The mode declarations are used in order to constrain the hypotheses' variables in the following ways:

  – An input variable of a given type in a body literal must appear as an input variable of the same type in the head literal, or as an output variable of the same type in an earlier body literal of that clause.

– An output variable of a given type in the head literal must appear as an output variable of the same type in some body literal of that clause.

Type and mode declarations received some attention in ILP research, like for example in [MS95] and [CKM$^+$05], exactly because of their great impact both on the hypothesis search space and the language bias and expressiveness. We will discuss the language bias of our system in Section 5.3.

5.2 Sampling

As we have already seen in Section 2.2.8, one of the main reasons why ILP is expensive is the fitness test of each explored clause with respect to the data, by checking for each example, whether or not it is subsumed by the clause. In knowledge bases, which are extracted from the web and can be really large this can be prohibitive for the application of an ILP algorithm. YAGO [SKW07], for example, contains about 20 million facts[1]. Nevertheless, in some cases, we do not need all examples-facts existing in a knowledge base in order to learn a “good” rule, but we can choose a representative sample from the available data and run the ILP algorithm only on these sample data.

In [Sri99], A. Srinivasan proposes two sampling methods for analyzing large datasets in the context of ILP. The first, subsampling, is a single-sample design in which the utility of a potential rule is evaluated on a randomly selected sub-sample of the data. The second, logical windowing, is a multiple-sample design that tests and sequentially includes errors made by a partially correct theory.

In *subsampling*, the algorithm is given an upper limit (say m) on the number of examples to be used in evaluating the best literal to select at each step (here we assume a hill climbing approach like Algorithm 2.1). Should more than this number of examples be present, a random subsample of m examples is selected. The literal in the body of the rule that best discriminates among classes (positive and negative examples) in this sample is selected. The procedure then continues with *all* examples (that is, not just the m chosen in the previous step) being passed to the next level. This last step is important to avoid being left with progressively fewer examples as the length of the clause increases.

Logical windowing stems from techniques developed to enable decision-tree methods in order to cope with large datasets. Here, a small sample (or window) of fixed size

[1]according to the official web page of the YAGO project http://www.mpi-inf.mpg.de/yago-naga/yago

is drawn from the examples. These form the training set used to construct the decision-tree. The tree is then tested on all examples. If this results in an intolerable number of errors, a small sample of the examples that are erroneously classified are added to the training set and an entirely new tree is re-learned.

Although both methods are very interesting and have been widely used and implemented by the ILP community, they have some limitations when considering an incomplete knowledge base and in the absence of explicit negative examples. In subsampling the sample population is not remaining the same, but in each step we need to resample randomly. In this way, there is no guarantee that support will preserve the anti-monotonicity property, so we miss the chance of early pruning. Apart from that, both subsampling and logical windowing presume an ILP algorithm, which uses beam search or hill climbing and is searching for a local optimum. In our case, we are interested in learning not the best rule for a given relation, but many rules that might be interesting. In this case, if we change the example set in each step, we are not able to compare the predictive strength of different rules. For example, if the rule

$$livesIn(x, y)\text{:-}bornIn(x, y), diedIn(x, y)$$

and the rule

$$livesIn(x, y)\text{:-}politicianOf(x, y), hasChild(x, z), bornIn(z, y)$$

are evaluated over different training sets, then we cannot compare their confidences anymore in order to decide which one is better. In addition, logical windowing requires the testing of the learned tree on all testing data. If we have not kept only one clause as the hypothesis, but many, we will have to test all of these clauses on all data. Moreover, the idea of relearning is not very appealing.

Subsampling and logical windowing might be efficient when we are trying to learn "hard" rules (formulas that hold always), when we can apply beam search or hill climbing and when we have negative examples in the training set. But when our aim is to learn soft rules (i.e., rules that express something that *usually* holds, but not always) and therefore, we cannot apply beam search or hill climbing, then these methods can be problematic.

In our setting, we would like to evaluate the rules for each target relation on a subset of the original data. In order to keep the anti-monotonicity property of support (which gives us opportunities for early pruning) and the confidence of the rules comparable, we need to sample once in the start of the learning as a preprocessing step and keep the sample the same until the end. Apart from that, we only have positive examples, and we make assumptions about the negative ones, as described in Sections 4.8 and 4.2. This means that if we sample on a per-fact basis we might introduce more incompleteness, by including in the sample only some of the facts related with a given entity, as described Section 4.8. For example, assume that we are trying to learn a rule about the relation $livesIn$, and we have two examples $livesIn(C, T)$ and $livesIn(C, S)$ from which we sample $livesIn(C, S)$. Then a rule that predicts correctly only $livesIn(C, T)$ will be unfairly penalized. In order to avoid the introduction of more incompleteness in our data set, we sample uniformly-randomly not according to facts, but according to entities of the input argument. This means that in the previous example, we will have to include both $livesIn(C, T)$ and $livesIn(C, S)$ in our training sample.

At this point, we need to make clear that, in our approach, we only sample the training examples in the head predicate and not the all the data of the background knowledge. If we try to sample also the background knowledge randomly, there is the danger that we will break the dependencies and connections that exist in reality between the entities.

5.3 Language Bias

Any mechanism employed by a learning system to constrain the search for hypothesis is named *bias* [UM82]. Declarative bias denotes explicit, user-specified bias, which can preferably be formulated as a modifiable parameter of the system. Bias can either determine how the hypothesis space is searched (*search bias*) or determine the hypothesis space itself (*language bias*). By selecting a stronger language bias (a less expressive hypothesis language) the search space becomes smaller and the learning more efficient. However, this might prevent the system from finding a solution which is not contained in the less expressive language [LD93].

As we have already seen in Section 5.1, existing ILP systems are using predicate and mode declarations as a language bias in order to make the ILP learning algorithm more efficient. In our setting, we also impose some language bias. To be more detailed, we do not allow the following kind of rules.

- Rules which have already been considered but with different order in the literals, e.g., $livesIn(x, y)$:-$bornIn(x, y), diedIn(x, y)$ vs $livesIn(x, y)$:-$diedIn(x, y), bornIn(x, y)$.
- Rules which contain in the body, the head literal itself with exactly identical variables, e.g., $livesin(x, y)$:-$livesIn(x, y)$.

- Rules which contain in the body, the head literal with the same input argument, under the condition that the arguments of the head relation are of different type, e.g., $livesIn(x,y)$:-$livesIn(x,z)$. However, if the arguments are of the same type, we allow such rules, because there might be a transitivity property in the head relation. For example, the rule $locatedIn(x,y)$:-$locatedIn(x,z)$ is constructed, because x and y are both of type *location* and the rule is an intermediate step for the rule $locatedIn(x,y)$:-$locatedIn(x,z), locatedIn(z,y)$ which expresses the transitivity property of the *locatedIn* relation.

- Rules which contain the same literal twice.

- Rules which violate the functional property of a relation, e.g., in the rule

$$livesIn(x,y)\text{:-}bornIn(x,y), bornIn(x,z)$$

  y and z will be the same in any case, because *bornIn* is a functional relation.

- Rules which bind a relation to itself but try to put a new variable in the same place where there was an old one. For example, we will not consider the rule $livesIn(x,y)$:-$bornIn(x,y), bornIn(z,y)$, where x,y are the old variables and z is introduced as a new variable.

In addition to the above guidelines for exploring the search space, the final rules that we are interested in producing, should be well-formed Datalog clauses. In other words, both the variables of the head should appear also in the body. If we want to restrict the search space more, we can require from a valid final rule not to have any free variable. In this way, we will avoid considering as final answers, rules like $livesIn(x,y)$:-$bornIn(x,y), hasChild(x,z)$, where z is a free variable. This restriction is possible in our implementation by setting a flag. In the case that we do not allow the existence of free (unbounded) variables in the final rules, we are able to restrict the branching factor of the search space even more. If we are given as input to our algorithm the length d of the rules that we are interested in creating, then in each step of the ILP algorithm, we add a literal only if we are sure that this literal will not end up with a free variable by the end of the learning procedure. For example, imagine that we are trying to learn for the relation $isMarriedTo(x,y)$ rules with bodies of length 2. Also imagine that if we use in the first step the relation *bornOnDate* the only way to include in the body both the variables x and y is if we construct the rule $isMarriedTo(x,y)$:-$bornOnDate(x,z), establishedOnDate(w,z), bornIn(y,w)$, which is a rule of length 3. In this case, the relation *bornOnDate* will not be considered as a candidate literal in the first step of the ILP algorithm, although the overlap of the entities of the first arguments

$politicianOf(x,y)$:-$bornIn(x,z), isLeaderOf(x,y), type(x, President)$
confidence: 0.65 support: 0.01 specialityRatio: 0.25
$N^+(c)$: 31 $E^+(c)$: 33 $N(c)$: 47

$politicianOf(x,y)$:-$bornIn(x,z), isCitizenOf(x,y), type(x, President)$
confidence: 1.0 support: 0.01 specialityRatio: 0.20
$N^+(c)$: 29 $E^+(c)$: 30 $N(c)$: 29

$politicianOf(x,y)$:-$bornIn(x,z), locatedIn(z,y), type(x, Governor)$
confidence: 0.51 support: 0.09 specialityRatio: 0.25
$N^+(c)$: 98 $E^+(c)$: 194 $N(c)$: 192

$politicianOf(x,y)$:-$bornIn(x,z), locatedIn(z,y), type(x, President)$
confidence: 0.65 support: 0.015 specialityRatio: 0.12
$N^+(c)$: 21 $E^+(c)$: 34 $N(c)$: 32

$politicianOf(x,y)$:-$bornIn(x,z), locatedIn(z,y), type(x, Chancellor)$
confidence: 0.56 support: 0.01 specialityRatio: 0.14
$N^+(c)$: 14 $E^+(c)$: 25 $N(c)$: 25

$politicianOf(x,y)$:-$bornIn(x,z), hasCapital(y,z), type(x, President)$
confidence: 0.57 support: 0.03 specialityRatio: 0.21
$N^+(c)$: 52 $E^+(c)$: 66 $N(c)$: 90

$politicianOf(x,y)$:-$bornIn(x,z), hasCapital(y,z), type(x, Chancellor)$
confidence: 0.37 support: 0.01 specialityRatio: 0.12
$N^+(c)$: 18 $E^+(c)$: 30 $N(c)$: 48

$politicianOf(x,y)$:-$diedIn(x,z), isCitizenOf(x,y), type(x, President)$
confidence: 1.0 support: 0.01 specialityRatio: 0.26
$N^+(c)$: 27 $E^+(c)$: 27 $N(c)$: 27

$politicianOf(x,y)$:-$diedIn(x,z), locatedIn(z,y), type(x, Governor)$
confidence: 0.82 support: 0.09 specialityRatio: 0.29
$N^+(c)$: 156 $E^+(c)$: 191 $N(c)$: 188

$politicianOf(x,y)$:-$diedIn(x,z), locatedIn(z,y), type(x, President)$
confidence: 0.58 support: 0.01 specialityRatio: 0.17
$N^+(c)$: 17 $E^+(c)$: 30 $N(c)$: 29

Table 5.1: Rules with free variables for the relation $politicianOf$.

of the relations $isMarriedTo$ and $bornOnDate$ might pass the thresholds of Algorithm 5.2.

$politicianOf(x, y)$:-$diedIn(x, z), hasCapital(y, z), type(x, President)$
confidence: 0.40 support: 0.04 specialityRatio: 0.24
$N^+(c)$: 59 $E^+(c)$: 94 $N(c)$: 145

$politicianOf(x, y)$:-$diedIn(x, z), hasCapital(y, z), type(x, Chancellor)$
confidence: 0.45 support: 0.01 specialityRatio: 0.18
$N^+(c)$: 29 $E^+(c)$: 41 $N(c)$: 64

$politicianOf(x, y)$:-$diedIn(x, z), hasCapital(y, z), type(x, Officeholder)$
confidence: 0.53 support: 0.01 specialityRatio: 0.11
$N^+(c)$: 15 $E^+(c)$: 25 $N(c)$: 28

$politicianOf(x, y)vdiedOnDate(x, z), isCitizenOf(x, y), type(x, President)$
confidence: 1.0 support: 0.01 specialityRatio: 0.24
$N^+(c)$: 33 $E^+(c)$: 33 $N(c)$: 33

$politicianOf(x, y)$:-$isAffiliatedTo(x, z), isCitizenOf(x, y), type(x, President)$
confidence: 1.0 support: 0.009 specialityRatio: 0.21
$N^+(c)$: 20 $E^+(c)$: 21 $N(c)$: 20

$politicianOf(x, y)$:-$isLeaderOf(x, y), politicianOf(z, y), (z \neq x)$
confidence: 0.98 support: 0.038 specialityRatio: 0.19
$N^+(c)$: 79 $E^+(c)$: 81 $N(c)$: 80

$politicianOf(x, y)$:-$isLeaderOf(x, y), type(x, president)$
confidence: 0.65 support: 0.02 specialityRatio: 0.26
$N^+(c)$: 43 $E^+(c)$: 46 $N(c)$: 66

$politicianOf(x, y)$:-$isLeaderOf(x, y), type(x, President), hasPredecessor(x, w)$
confidence: 0.72 support: 0.02 specialityRatio: 0.25
$N^+(c)$: 39 $E^+(c)$: 42 $N(c)$: 54

$politicianOf(x, y)$:-$isLeaderOf(x, y), type(x, President), isCitizenOf(w, y)$
confidence: 1.0 support: 0.01 specialityRatio: 0.32
$N^+(c)$: 33 $E^+(c)$: 34 $N(c)$: 33

TABLE 5.2: Rules with free variables (continuation of Table 5.1) for the relation $politicianOf$.

$politicianOf(x,y)$:-$isLeaderOf(x,y), type(x,President), isLeaderOf(w,y), (x \neq w)$
confidence: 0.84 support: 0.018 specialityRatio: 0.26
$N^+(c)$: 37 $E^+(c)$: 38 $N(c)$: 44

$politicianOf(x,y)$:-$isLeaderOf(x,y), type(x,President), originatesFrom(w,y)$
confidence: 1.0 support: 0.01 specialityRatio: 0.26
$N^+(c)$: 24 $E^+(c)$: 24 $N(c)$: 24

$politicianOf(x,y)$:-$isLeaderOf(x,y), type(x,President), happenedIn(w,y)$
confidence: 1.0 support: 0.013 specialityRatio: 0.26
$N^+(c)$: 27 $E^+(c)$: 28 $N(c)$: 27

$politicianOf(x,y)$:-$isLeaderOf(x,y), type(x,Chancellor)$
confidence: 0.74 support: 0.01 specialityRatio: 0.20
$N^+(c)$: 23 $E^+(c)$: 24 $N(c)$: 31

$politicianOf(x,y)$:-$hasPredecessor(x,z), isCitizenOf(x,y)$
confidence: 0.93 support: 0.023 specialityRatio: 0.10
$N^+(c)$: 46 $E^+(c)$: 51 $N(c)$: 49

$politicianOf(x,y)$:-$hasPredecessor(x,z), isCitizenOf(z,y)$
confidence: 0.97 support: 0.01 specialityRatio: 0.12
$N^+(c)$: 39 $E^+(c)$: 40 $N(c)$: 40

$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$
confidence: 0.98 support: 0.37 specialityRatio: 0.28
$N^+(c)$: 774 $E^+(c)$: 796 $N(c)$: 788

$politicianOf(x,y)$:-$hasSuccessor(x,z), isCitizenOf(x,y)$
confidence: 0.97 support: 0.02 specialityRatio: 0.11
$N^+(c)$: 44 $E^+(c)$: 45 $N(c)$: 45

$politicianOf(x,y)$:-$hasSuccessor(x,z), isCitizenOf(z,y)$
confidence: 0.94 support: 0.01 specialityRatio: 0.11
$N^+(c)$: 36 $E^+(c)$: 39 $N(c)$: 38

$politicianOf(x,y)$:-$hasSuccessor(x,z), isLeaderOf(z,y)$
confidence: 0.65 support: 0.024 specialityRatio: 0.11
$N^+(c)$: 44 $E^+(c)$: 50 $N(c)$: 67

Table 5.3: Rules with free variables (continuation of Table 5.1) for the relation *politicianOf*.

$politicianOf(x,y)$:-$hasSuccessor(x,z), politicianOf(z,y), (z \neq x)$
confidence: 0.97 support: 0.37 specialityRatio: 0.28
$N^+(c)$: 749 $E^+(c)$: 774 $N(c)$: 771

$politicianOf(x,y)$:-$familyNameOf(z,x), isCitizenOf(x,y), type(x, President)$
confidence: 1.0 support: 0.01 specialityRatio: 0.21
$N^+(c)$: 38 $E^+(c)$: 40 $N(c)$: 38

$politicianOf(x,y)$:-$givenNameOf(z,x), isCitizenOf(x,y), type(x, President)$
confidence: 1.0 support: 0.018 specialityRatio: 0.21
$N^+(c)$: 38 $E^+(c)$: 40 $N(c)$: 38

$politicianOf(x,y)$:-$hasPredecessor(z,x), isCitizenOf(z,y)$
confidence: 0.88 support: 0.02 specialityRatio: 0.11
$N^+(c)$: 46 $E^+(c)$: 53 $N(c)$: 52

$politicianOf(x,y)$:-$hasPredecessor(z,x), isLeaderOf(z,y)$
confidence: 0.72 support: 0.03 specialityRatio: 0.11
$N^+(c)$: 62 $E^+(c)$: 69 $N(c)$: 86

$politicianOf(x,y)$:-$hasPredecessor(z,x), politicianOf(z,y), (z \neq x)$
confidence: 0.97 support: 0.34 specialityRatio: 0.28
$N^+(c)$: 693 $E^+(c)$: 711 $N(c)$: 713

$politicianOf(x,y)$:-$hasPredecessor(z,x), isCitizenOf(x,y), type(x, President)$
confidence: 1.0 support: 0.01 specialityRatio: 0.23
$N^+(c)$: 26 $E^+(c)$: 26 $N(c)$: 26

$politicianOf(x,y)$:-$hasSuccessor(z,x), isCitizenOf(z,y)$
confidence: 1.0 support: 0.021 specialityRatio: 0.11
$N^+(c)$: 45 $E^+(c)$: 47 $N(c)$: 45

$politicianOf(x,y)$:-$hasSuccessor(z,x), politicianOf(z,y), (z \neq x)$
confidence: 0.96 support: 0.34 specialityRatio: 0.28
$N^+(c)$: 691 $E^+(c)$: 717 $N(c)$: 717

$politicianOf(x,y)$:-$hasSuccessor(z,x), isCitizenOf(x,y)$
confidence: 1.0 support: 0.01 specialityRatio: 0.11
$N^+(c)$: 35 $E^+(c)$: 36 $N(c)$: 35

TABLE 5.4: Rules with free variables (continuation of Table 5.1) for the relation *politicianOf*.

5.4 The Main Algorithm

In this section, we will present shortly our main algorithm for creating soft rules in the form of function-free Horn Clauses, which are taking into account the language bias as it is described in Section 5.3.

5.4.1 Search Strategy

As we have already seen in Section 2.2.7, the majority of ILP systems apply an informed search strategy, in order to avoid searching exhaustively the hypothesis space. In each step, the clause needs to be refined by adding a new literal to its body . The algorithm evaluates all candidate literals according to some heuristics (accuracy, accuracy gain, weighted accuracy gain, information gain or weighted accuracy gain [LD93], or relational information gain [LJFP10]), and only the refined rule with the "best" literal is kept (if we apply hill climbing) or the refined rules with the best b literals are kept (if we apply beam search with beam width b). An empirical comparison of search heuristics ([Min89]) showed that there is little difference between them and that their use more reduces the size than it improves the accuracy of induced hypotheses. Such heuristics do not have a monotonic behavior, so in practice their use leads to a trade-off between efficiency and completeness. The algorithm finds clauses faster, but there is no guarantee that it will find all the good clauses, or that the clauses that it will induce will be a global optimum.

In our setting, we are interested in learning soft rules. That is, rules that might not hold always, but they hold with some probability. In other words we are interested not only in finding the best rules, but also rules that might not be "so good". If we apply an informed search strategy, we might skip such interesting rules. For this reason, we apply exhaustive search in the search space. Nevertheless, we do give the option to the user (by setting a flag) to switch to a beam search version (based on a confidence gain heuristic) of our algorithm.

5.4.2 Pruning Strategy

As we have already seen in Section 4.6, the only quantities that monotonically grow smaller during the ILP refinement process are the positive examples covered ($N^+(c)$), the possible positive examples to be covered ($E^+(c)$) and the *support*. Therefore, will we allow the pruning of branches in the hypothesis space only if those quantities fall below some user defined thresholds. On the other hand, the measure of confidence is not behaving monotonic, so we cannot use it for early pruning, in the normal case.

Only if the user chooses to use beam search, we stop expanding rules that have a lower confidence than their parent rules (applying a confidence-gain heuristic). Otherwise, we keep expanding the rule with the hope that in the next steps we will get a rule with higher confidence. At the same time, we do not keep such rules as a possible output of the algorithm. For example, imagine that we have the rule:

$livesIn(x, y)$:-$bornIn(x, y)$ with confidence 0.60

and the refined rule of the next step is:

$livesIn(x, y)$:-$bornIn(x, y), diedIn(x, y)$ with confidence 0.50

According to our approach, we will continue expanding the second rule with the hope of getting a better clause in the next steps, but the second clause itself is of no interest for us, because we have already added the first rule to the set of output rules, which covers all examples that the second covers (and perhaps even more), and at the same time has a better confidence.

5.4.3 Speciality Threshold and Type Refinements

As described in Section 4.5, apart from support and confidence, we also need the measure of speciality ratio in order to avoid the generation of overly general rules. In our approach, we keep a threshold also for the speciality ratio and if the rule's speciality ratio is lower than this threshold, the rule is marked as too general. This means that the rule itself will not be added to the output of the algorithm, but we will continue to refine it in the hope that we will get a rule that is more specific in the next iterations.

In the case that the current rule is found to be overly general, we also apply the following strategy of imposing more refinements to the clause:

- Before starting the learning process for the current target predicate, we find the n most popular types of the entities of the target's input argument. For example, if we try to learn the predicate $politicianOf$ and the input argument is the first argument, then the $n = 3$ most popular types in YAGO for the entities of the first argument are: $Governor$, $President$ and $Chancellor$. Figure 5.2 shows the SQL statement for finding the above types for this example.
- Once we find a rule that is overly general, we investigate the rules that are produced from this rule just by imposing a type restriction in the argument of the *body*, according to the types we have found in the previous step. For example, if the

rule $politicianOf(x, y)$:-$bornIn(x, y)$ is found to be too general, we will investigate how the rules:

$politicianOf(x, y)$:-$bornIn(x, y), type(x, Governor)$

$politicianOf(x, y)$:-$bornIn(x, y), type(x, President)$

$politicianOf(x, y)$:-$bornIn(x, y), type(x, Chancellor)$

behave with respect to support, confidence and speciality ratio.

In Tables 5.5- 5.11 we can see some examples of such rules.

```
SELECT arg2
FROM
        (SELECT f1.arg2, count(*) c
         FROM facts f0, facts f1
        WHERE f0.relation='politicianOf'  AND  f1.relation='type'
        AND  f0.arg1=f1.arg1
        GROUP BY f1.arg2
        ORDER BY c DESC)
WHERE rownum<=3;
```

FIGURE 5.2: Finding popular types for the first argument of the relation $politicianOf$.

5.4.4 Constants

Apart from constructing rules that contain only variables, we are also interested in constructing rules which contain constants. The user can enable the learning-with-constants-mode of the algorithm by setting a flag. In this case, for every rule which passes the thresholds of support, positive examples covered ($N^+(c)$) and possible positive examples to be covered ($E^+(c)$), we investigate if there are constants that might produce good rules. The existing algorithm tries to substitute with constants only the variables that exist in the head of the rule. That is, if we have the rule $politicianOf(x, y)$:-$hasPredecessor(x, z), politicianOf(z, y)$, we will only search for the constants for the variables x and y that exist in the *head*, but not for z. The reason for this decision is the high cost, which a search for constants for new variables will impose to our algorithm. In other words, we are in general trying to keep a trade-off between the expressivity of the description language we are using and the high computational cost, which this implies. Apart from that, it is obvious that the use of constants for specific arguments of specific relations does not make sense. For example, for the relation $politicianOf(x, y)$, we would expect that we can get a good rule by using a constant for the second argument (learn a rule for the politicians of a specific country), but not for the first. We assume that the user gives as input the information of which argument of the target relation can take a constant.

For each rule, we are investigating if there are any constants that can form rules that can possibly pass the thresholds. If there are, we form new rules that include these constants and evaluate them. Figure 5.3 shows the SQL statement for finding constants for the variable y of the rule $politician(x, y)$:-$born(x, z), locatedIn(z, y)$ that can possibly lead to the construction of rules that will pass the thresholds of support, positive examples covered ($N^+(c)$) and possible positive examples to be covered ($E^+(c)$). Tables 5.5- 5.11 show the induced rules with constants for different target predicates. The parameters for learning were: rule length: 2, confidence threshold: 0.25, support threshold: 0.05, possible positive examples to be covered ($E^+(c)$) threshold: 10, positive examples covered ($N^+(c)$) threshold: 10, speciality ratio threshold: 0.10. Notice that for some target predicates like $graduatedFrom$, we are not able to learn any rule with constants. Also notice that these results correspond to relatively low thresholds of positive examples covered ($N^+(c)$) and possible positive examples to be covered ($E^+(c)$). The higher these thresholds, the fewer rules with constants will be produced.

5.4.5 Algorithmic Framework

In this section, we will put all the previous ideas together and summarize our inductive learning algorithm. In order to keep things as simple as possible, we will ignore side-versions of the algorithm like beam search and the option to use constants.

As input of our algorithm we assume:

- A target predicate that we are interested in learning.
- A maximum body length *depth* that the induced rules will have.
- The input argument of the target relation .
- The background knowledge in the form of a database table $facts$ of the form (relation,arg1,arg2).
- The positive training examples in the form of a database table *train* of the form (relation,arg1,arg2) which are a result of the sampling procedure, described in Section 5.2.

```sql
-- filter the constants that pass the threshold for positives covered
SELECT arg2
FROM
        ( SELECT  f1.arg1 , f2.arg2
          FROM train11 f0, facts f1, facts f2
          WHERE  f0.relation='politicianOf'  AND  f1.relation='bornIn' AND f2.relation='locatedIn'
          AND  f0.arg1=f1.arg1  AND  f0.arg2=f2.arg2  AND  f2.arg1=f1.arg2
          GROUP BY  f1.arg1, f2.arg2)
GROUP BY arg2
HAVING count(*)>thresPositivesCovered
INTERSECT
-- filter the constants that pass the thresholds for  possible positives to be covered and
--for support
SELECT arg2
FROM
        ( SELECT f0.arg1 input,f0.arg2
          FROM train11 f0
          WHERE f0.relation='politicianOf' AND f0.arg1 IN
                    (SELECT f1.arg1
                     FROM facts f1, facts f2
                     WHERE f1.relation='bornIn' AND f2.relation='locatedIn'
                      AND f2.arg1=f1.arg2 )
          GROUP BY f0.arg1,f0.arg2)
GROUP BY arg2
HAVING count(*)>thresPossiblePositivesToBeCovered OR count(*)>thresSupport*HeadSize
```

FIGURE 5.3: Finding constants for the variable y of the rule politician(x,y):-born(x,z), locatedIn(z,y).

RELATION TO BE LEARNED: politicianOf
$politicianOf(x,y)\textbf{:-}bornIn(x,z), locatedIn(z,y), type(x,Governor)$ confidence: 0.51 support: 0.09 specialityRatio: 0.25 $N^+(c)$: 98 $E^+(c)$: 194 $N(c)$: 192
$politicianOf(x,y)\textbf{:-}bornIn(x,z), locatedIn(z,y), type(x,President)$ confidence: 0.65 support: 0.01 specialityRatio: 0.12 $N^+(c)$: 21 $E^+(c)$: 34 $N(c)$: 32
$politicianOf(x,y)\textbf{:-}bornIn(x,z), locatedIn(z,y), type(x,Chancellor)$ confidence: 0.56 support: 0.01 specialityRatio: 0.14 $N^+(c)$: 14 $E^+(c)$: 25 $N(c)$: 25
$politicianOf(x,y)\textbf{:-}bornIn(x,z), hasCapital(y,z), type(x,President)$ confidence: 0.57 support: 0.03 specialityRatio: 0.21 $N^+(c)$: 52 $E^+(c)$: 66 $N(c)$: 90
$politicianOf(x,y)\textbf{:-}bornIn(x,z), hasCapital(y,z), type(x,Chancellor)$ confidence: 0.37 support: 0.01 specialityRatio: 0.12 $N^+(c)$: 18 $E^+(c)$: 30 $N(c)$: 48
$politicianOf(x,y)\textbf{:-}diedIn(x,z), locatedIn(z,y), type(x,Governor)$ confidence: 0.82 support: 0.09 specialityRatio: 0.29 $N^+(c)$: 156 $E^+(c)$: 191 $N(c)$: 188
$politicianOf(x,y)\textbf{:-}diedIn(x,z), locatedIn(z,y), type(x,President)$ confidence: 0.58 support: 0.01 specialityRatio: 0.17 $N^+(c)$: 17 $E^+(c)$: 30 $N(c)$: 29
$politicianOf(x,y)\textbf{:-}diedIn(x,z), hasCapital(y,z), type(x,President)$ confidence: 0.40 support: 0.04 specialityRatio: 0.24 $N^+(c)$: 59 $E^+(c)$: 94 $N(c)$: 145
$politicianOf(x,y)\textbf{:-}diedIn(x,z), hasCapital(y,z), type(x,Chancellor)$ confidence: 0.45 support: 0.01 specialityRatio: 0.18 $N^+(c)$: 29 $E^+(c)$: 41 $N(c)$: 64

Table 5.5: Rules learned for the $politicianOf$ relation.

RELATION TO BE LEARNED: politicianOf
$politicianOf(x,y)$**:-**$diedIn(x,z), hasCapital(y,z), type(x, Officeholder)$ confidence: 0.53 support: 0.01 specialityRatio: 0.11 $N^+(c)$: 15 $E^+(c)$: 25 $N(c)$: 28
$politicianOf(x,y)$**:-**$isLeaderOf(x,y), type(x, President)$ confidence: 0.65 support: 0.02 specialityRatio: 0.2 $N^+(c)$: 43 $E^+(c)$: 46 $N(c)$: 66
$politicianOf(x,y)$**:-**$isLeaderOf(x,y), type(x, Chancellor)$ confidence: 0.74 support: 0.01 specialityRatio: 0.20 $N^+(c)$: 23 $E^+(c)$: 24 $N(c)$: 31
$politicianOf(x,y)$**:-**$hasPredecessor(x,z), isCitizenOf(z,y)$ confidence: 0.97 support: 0.01 specialityRatio: 0.12 $N^+(c)$: 39 $E^+(c)$: 40 $N(c)$: 40
$politicianOf(x,y)$**:-**$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.98 support: 0.37 specialityRatio: 0.28 $N^+(c)$: 774 $E^+(c)$: 796 $N(c)$: 788
$politicianOf(x, France)$**:-**$hasPredecessor(x,z), politicianOf(z, France), (z \neq x)$ confidence: 1.0 support: 0.01 specialityRatio: 0.30 $N^+(c)$: 31 $E^+(c)$: 31 $N(c)$: 31

Table 5.6: Rules (continuation of Table 5.5) learned for the *politicianOf* relation.

RELATION TO BE LEARNED: politicianOf
$politicianOf(x, Massachusetts)$**:-**$hasPredecessor(x, z),$ $politicianOf(z, Massachusetts), (z \neq x)$ confidence: 1.0 support: 0.01 specialityRatio: 0.22 $N^+(c)$: 21 $E^+(c)$: 22 $N(c)$: 21
$politicianOf(x, y)$**:-**$hasSuccessor(x, z), isCitizenOf(z, y)$ confidence: 0.94 support: 0.01 specialityRatio: 0.11 $N^+(c)$: 36 $E^+(c)$: 39 $N(c)$: 38
$politicianOf(x, y)$**:-**$hasSuccessor(x, z), isLeaderOf(z, y)$ confidence: 0.65 support: 0.02 specialityRatio: 0.11 $N^+(c)$: 44 $E^+(c)$: 50 $N(c)$: 67
$politicianOf(x, y)$**:-**$hasSuccessor(x, z), politicianOf(z, y), (z \neq x)$ confidence: 0.97 support: 0.37 specialityRatio: 0.28 $N^+(c)$: 749 $E^+(c)$: 774 $N(c)$: 771
$politicianOf(x, France)$**:-**$hasSuccessor(x, z), politicianOf(z, France), (z \neq x)$ confidence: 1.0 support: 0.01 specialityRatio: 0.26 $N^+(c)$: 27 $E^+(c)$: 27 $N(c)$: 27
$politicianOf(x, y)$**:-**$hasPredecessor(z, x), isCitizenOf(z, y)$ confidence: 0.88 support: 0.02 specialityRatio: 0.12 $N^+(c)$: 46 $E^+(c)$: 53 $N(c)$: 52
$politicianOf(x, y)$**:-**$hasPredecessor(z, x), isLeaderOf(z, y)$ confidence: 0.72 support: 0.03 specialityRatio: 0.11 $N^+(c)$: 62 $E^+(c)$: 69 $N(c)$: 86
$politicianOf(x, y)$**:-**$hasPredecessor(z, x), politicianOf(z, y), (z \neq x)$ confidence: 0.97 support: 0.34 specialityRatio: 0.28 $N^+(c)$: 693 $E^+(c)$: 711 $N(c)$: 713
$politicianOf(x, France)$**:-**$hasPredecessor(z, x), politicianOf(z, France), (z \neq x)$ confidence: 1.0 support: 0.01 specialityRatio: 0.31 $N^+(c)$: 26 $E^+(c)$: 26 $N(c)$: 26
$politicianOf(x, y)$**:-**$hasSuccessor(z, x), isCitizenOf(z, y)$ confidence: 1.0 support: 0.02 specialityRatio: 0.11 $N^+(c)$: 45 $E^+(c)$: 47 $N(c)$: 45
$politicianOf(x, y)$**:-**$hasSuccessor(z, x), politicianOf(z, y), (z \neq x)$ confidence: 0.96 support: 0.34 specialityRatio: 0.28 $N^+(c)$: 691 $E^+(c)$: 717 $N(c)$: 717

TABLE 5.7: Rules (continuation of Table 5.6) learned for the *politicianOf* relation.

RELATION TO BE LEARNED: diedIn
$diedIn(x, London)$**:-**$bornIn(x, London)$ confidence: 0.37 support: 0.01 specialityRatio: 0.13 $N^+(c)$: 23 $E^+(c)$: 62 $N(c)$: 62
$diedIn(x, Paris)$**:-**$bornIn(x, Paris)$ confidence: 0.40 support: 0.005 specialityRatio: 0.15 $N^+(c)$: 11 $E^+(c)$: 27 $N(c)$: 27
$diedIn(x, y)$**:-**$isCitizenOf(x, z), hasCapital(z, y), type(x, Scientist)$ confidence: 0.25 support: 0.03 specialityRatio: 0.15 $N^+(c)$: 45 $E^+(c)$: 175 $N(c)$: 175
$diedIn(x, y)$**:-**$isCitizenOf(x, z), hasCapital(z, y), type(x, President)$ confidence: 0.59 support: 0.008 specialityRatio: 0.20 $N^+(c)$: 22 $E^+(c)$: 37 $N(c)$: 37
$diedIn(x, y)$**:-**$politicianOf(x, z), hasCapital(z, y), type(x, Person)$ confidence: 0.37 support: 0.008 specialityRatio: 0.12 $N^+(c)$: 14 $E^+(c)$: 37 $N(c)$: 37
$diedIn(x, y)$**:-**$politicianOf(x, z), hasCapital(z, y), type(x, Officeholder)$ confidence: 0.37 support: 0.008 specialityRatio: 0.15 $N^+(c)$: 15 $E^+(c)$: 39 $N(c)$: 40
$diedIn(x, y)$**:-**$politicianOf(x, z), hasCapital(z, y), type(x, President)$ confidence: 0.54 support: 0.01 specialityRatio: 0.13 $N^+(c)$: 46 $E^+(c)$: 85 $N(c)$: 85
$diedIn(x, y)$**:-**$hasPredecessor(x, z), diedIn(z, y), (z \neq x)$ confidence: 0.25 support: 0.126 specialityRatio: 0.21 $N^+(c)$: 148 $E^+(c)$: 573 $N(c)$: 573

TABLE 5.8: Rules learned for the *diedIn* relation.

RELATION TO BE LEARNED: livesIn
$livesIn(x,y)$**:-**$bornIn(x,y), type(x, Politician)$ confidence: 0.47 support: 0.04 specialityRatio: 0.10 $N^+(c)$: 112 $E^+(c)$: 235 $N(c)$: 235
$livesIn(x, Germany)$**:-**$hasAcademicAdvisor(x,z), isCitizenOf(z, Germany)$ confidence: 0.78 support: 0.005 specialityRatio: 0.12 $N^+(c)$: 22 $E^+(c)$: 28 $N(c)$: 28
$livesIn(x,y)$**:-**$hasAcademicAdvisor(x,z), livesIn(z,y), (z \neq x)$ confidence: 0.76 support: 0.006 specialityRatio: 0.11 $N^+(c)$: 26 $E^+(c)$: 34 $N(c)$: 34
$livesIn(x, Germany)$**:-**$hasAcademicAdvisor(x,z), livesIn(z, Germany), (z \neq x)$ confidence: 0.86 support: 0.004 specialityRatio: 0.13 $N^+(c)$: 19 $E^+(c)$: 22 $N(c)$: 22

Table 5.9: Rules learned for the *livesIn* relation.

RELATION TO BE LEARNED: bornIn
$bornIn(x, Paris)$**:-**$isCitizenOf(x,z), hasCapital(z, Paris)$ confidence: 0.25 support: 0.008 specialityRatio: 0.21 $N^+(c)$: 25 $E^+(c)$: 97 $N(c)$: 97
$bornIn(x,y)$**:-**$livesIn(x,y), type(x, Politician)$ confidence: 0.36 support: 0.02 specialityRatio: 0.26 $N^+(c)$: 95 $E^+(c)$: 263 $N(c)$: 263
$bornIn(x,y)$**:-**$livesIn(x,y), type(x, Officeholder)$ confidence: 0.33 support: 0.005 specialityRatio: 0.24 $N^+(c)$: 23 $E^+(c)$: 69 $N(c)$: 69
$bornIn(x,y)$**:-**$hasChild(z,x), bornIn(z,y), (z \neq x)$ confidence: 0.26 support: 0.006 specialityRatio: 0.12 $N^+(c)$: 26 $E^+(c)$: 83 $N(c)$: 100
$bornIn(x,y)$**:-**$hasChild(z,x), diedIn(z,y)$ confidence: 0.32 support: 0.004 specialityRatio: 0.10 $N^+(c)$: 18 $E^+(c)$: 51 $N(c)$: 55

Table 5.10: Rules learned for the *bornIn* relation.

RELATION TO BE LEARNED: graduatedFrom
$graduatedFrom(x,y)$**:-**$hasAcademicAdvisor(x,z), graduatedFrom(z,y), (z \neq x)$ confidence: 0.32 support: 0.09 specialityRatio: 0.21 $N^+(c)$: 52 $E^+(c)$: 159 $N(c)$: 159
$graduatedFrom(x,y)$**:-**$hasAcademicAdvisor(x,z), worksAt(z,y)$ confidence: 0.51 support: 0.06 specialityRatio: 0.2 $N^+(c)$: 55 $E^+(c)$: 105 $N(c)$: 107

Table 5.11: Rules learned for the *graduatedFrom* relation.

Parameter	**Threshold**
confidence	confThres
support	suppThres
positive examples covered n^+	nThres
possible positive examples to be covered E^+	EThres
speciality ratio	specThres

Table 5.12: Parameters and Thresholds

- The n most popular types of the input argument of the target relation, which are found as described in Section 5.4.3.
- Four hash map structures which map every relation to a list of relations that the first relation can be joined on specific arguments, which are a result of the preprocessing step described in Section 5.1. arg1JoinOnArg1 will denote the structure that contains the mappings from relations to lists of relations that their first argument can be joined with the first argument of the key relation and so on.
- Thresholds for support, confidence, speciality ratio, positive examples covered ($N^+(c)$), possible positive examples to be covered ($E^+(c)$). Table 5.12 shows the names of the measures and the names of their corresponding thresholds, which will be used in the algorithms later on.

As we iterate through the ILP algorithm, we store the produced rules in a tree structure, which has as root the empty clause $target(x, y) \leftarrow \emptyset$. As we proceed with the ILP algorithm, more rules are constructed from previous rules and therefore more nodes are added in the rule tree. Algorithms 5.3, 5.4 and 5.5 offer a high level description of the algorithm. In Algorithm 5.3, we try to extend the rule by adding a new literal that can be joined with the head literal first and then we repeat for each literal in the body. Algorithm 5.4 shows how we construct the new rule by constructing and adding a new literal in the body. Method *checkValidityOfRule()* checks if the rule is a valid rule according to the language bias described in Section 5.3. Method *getJoinableRelations(joinCase,l.getRelation())* returns all relations, which can be joined with the relation of literal l in a way implied by the $joinCase$ parameter, based on the hash maps calculated in the preprocessing step of Section 5.1. Method *extendWithTypes(newRule)* forces further refinement to the rule according to types as discussed in Section 5.4.3. Algorithm 5.5 describes all the threshold checking taking place for the new rule, before we decide that this rule should be stored in the rule-tree structure or not. Algorithm 5.3 then continues by extending each new rule, which is just constructed in a similar fashion. In the end, we only have to parse the rule-tree structure and output the rules, which are marked as good.

Algorithm 5.3: The extendRule method

```
Data: node: a RuleTreeNode
d: the current depth in the tree

// refine head literal
1  if inputArg=1 then
       // start by binding on 1st arg of head
2
       // 1st arg of head on 1st arg of some relation
3      refinePredicate(node,d,1,node.getHeadLiteral)
       // 1st arg of head on 2nd arg of some relation
4      refinePredicate(node,d,2,node.getHeadLiteral)
5  else
       // start by binding on 2nd arg of head
6
       // 2nd arg of head on 1st arg of some relation
7      refinePredicate(node,d,3,node.getHeadLiteral)
       // 2nd arg of head on 2nd arg of some relation
8      refinePredicate(node,d,4,node.getHeadLiteral)
9
   // refine body literals
10 foreach Literal l in the body of the rule do
11     refinePredicate(node,d,1,l)
12     refinePredicate(node,d,2,l)
13     refinePredicate(node,d,3,l)
14     refinePredicate(node,d,4,l)
15
16 if node.getChildren().size() > 0 and d < depth then
17
       // extend every child-rule created
18     foreach node child in node.getChildren() do
19         extendRule(child,d+1);
```

Algorithm 5.4: The refinePredicate method

Data: *node*: a RuleTreeNode
d: the current depth in the tree
joinCase: a way of joining between the original literal *l* and a candidate relation. e.g., 1st argument of *l* with 1st argument of candidate relation
l: the literal of the body on which the join will happen

```
candidateRelations:=getJoinableRelations(joinCase, l.getRelation())

foreach relation in candidateRelations do
    create a newLiteral

    // create a new rule by adding the new literal
    newRule:=node.getRule().clone()
    newRule.addLiteral(newLiteral)

    if checkValidityOfRule(newRule) then

        if evaluateRule(node,d,newRule) then
            if newRule.isTooGeneral then
                extendWithTypes(newRule)
```

Algorithm 5.5: The evaluateRule method.

```
Data: node: a RuleTreeNode
d: the current depth in the tree
newRule: the new rule which we want to evaluate

Result: true if newRule is stored, false otherwise

calculate support, $E^+(newRule)$
if $newRule.getSupport < suppThres$ or $newRule.getE^+ < EThres$) then
    return false

calculate confidence, $n^+(newRule)$, specialityRatio
if $newRule.getSpecialityRatio < specThres$ then
    newRule.isTooGeneral:=true

if $newRule.getConfidence > confThres$ and $newRule.get_n^+ > nThres$ then
    newRule.isGood:=true

if newRule.isGood then

    if confidenceGain(newRule, newRule.getBestAncestor())>0 then
        newRule.isGood:=true
    else
        newRule.isGood:=false

    node.addChild(newRule)

    return true
return false
```

Chapter 6

Experimental Evaluation and Results

Work on this thesis was developed as a part of the URDF project for efficient reasoning in uncertain RDF knowledge bases with soft and hard rules, described in Section 3. In this chapter, we present the results of reasoning, in terms of precision and recall, when we use as soft rules, the rules outputted from our rule learner. Practically, we evaluate the quality the rules learned, by measuring their deductive power: we report the precision of the facts derived by any rule for each relation, for which there exist a rule with this relation as head predicate, separately. We also report the number of new facts produced for each such relation. We conduct experiments for two different settings: one for propositional reasoning by using the URDF framework, described in Section 6.3 and one for probabilistic reasoning by using the software called Alchemy and its underlying theory of Markov Logic Networks, presented in Section 6.4.

6.1 The Data Set

As a data set for learning rules we used YAGO. YAGO [SKW07](Yet Another Great Ontology) is an ontology automatically extracted from Wikipedia and WordNet. It contains 20 million RDF-facts about more than 2 million entities. YAGO has a manually confirmed accuracy of 95%.

6.2 Evaluation Measures

As it has already been mentioned, the quality of the derived facts for each relation was measured in terms of precision. Precision is defined in our case as:

$$Precision = \frac{true\,derived\,facts}{derived\,facts} \tag{6.1}$$

For measuring precision, we took a random sample of 700 facts, and we assessed the precision in this sample. We also estimated the Wilson interval[Wil27] for $\alpha = 5\%$. The evaluation of the sample was done manually; for each fact, we checked the corresponding Wikipedia page (YAGO contains facts extracted from Wikipedia) and if the fact was stated there either in textual form or in an info-box, it was assessed as true, otherwise as false. In other words, we are using Wikipedia as a ground truth for our experiments. Note also that precision is defined in Equation 6.1 in a very similar way to the accuracy of a clause:

$$Accuracy(c) = \frac{positive\,examples\,covered\,by\,c}{examples\,covered\,by\,c} \tag{6.2}$$

Nevertheless, we should not forget that accuracy is defined for a clause c alone, whereas in our experiments we measure the precision in an actual reasoning setting, i.e., we measure the precision of the facts derived by *all* rules. And not only by all rules of the same head predicate, but by all rules in general, which means that if there is recursion between the rules, this will surely affect the precision, because even more facts will be generated. Apart from that, in our reasoning setting in Section 6.3, we also resolve the possible inconsistencies produced by the rules, by means of the special Max-Sat solver of URDF, which also affects the precision of the facts return to the user as an answer. So, in the end we do not measure the accuracy of a clause alone, but the precision of the facts that a *set* of rules altogether can return, in order to satisfy a specific information need (query) of the user, within a reasoning framework which is able to resolve inconsistencies.

Another measure that is usually used in information retrieval is recall. Recall is defined in our case as:

$$Recall = \frac{true\,derived\,facts}{all\,true\,facts} \tag{6.3}$$

and it can be seen as the probability that a true fact will be produced by some of our rules. Unfortunately, in our case, we do not know how many true facts exist in general, which makes the calculation of recall infeasible. For this reason, instead of recall, we report, for each predicate, both the number of facts initially included in our database without any rules, and the number of facts, both original and deduced, which exist in the knowledge base with the learned soft rules.

6.3 URDF Experiments

6.3.1 URDF: Experimental Setting

The setting for this experiment is in reality a propositional reasoning setting. In propositional reasoning we present to the user as an answer only the most consistent view (possible world) over a potentially inconsistent knowledge base. For this reason we make use of the URDF framework, which includes an efficient MAX-SAT approximation algorithm for resolving inconsistencies.

In our learning setting, we partitioned the YAGO knowledge base into 3 partitions, by applying appropriate sampling as it is described in Section 5.2, and we ran the learning procedure for each partition separately. The main idea of the experiment is to learn rules for different confidence thresholds (0.90, 0.85, 0.50, 0.25) and then measure the precision of the derived facts.

After generating rules for all predicates of interest for a specific confidence threshold, we import the rules as soft rules in URDF's reasoner, and we run queries for each predicate, e.g., $politicianOf(?x, ?y)$, in order to get all possible facts that either exist in the original database or can be derived by the rules. In the end, we measure the precision of the facts, which are produced by some rule.

In this way, we tried to resemble a cross-validation-like setting; we partition the knowledge base in three partitions, so that we will learn rules from different training data, but with the same background knowledge. Then, we test the quality of the derived facts in terms of average precision and average number of the facts produced for each partition, by using the whole knowledge base. In other words, if $HeadPredicates(i)$ is the set of the head predicates of all rules in a rule-set learned for the partition i, p is a predicate, k_p is the number of partitions for which $p \in HeadPredicates$ and $Precision(p, i)$ is the precision achieved for the query $p(?x, ?y)$ in the i partition, then the reported precision is:

$$Average\ Precision(p) = \frac{\sum_{i=1}^{k_p} Precision(p, i)}{k_p}$$

Relations	$politicianOf$, $livesIn$, $bornIn$, $actedIn$, $familyNameOf$, $hasPredecessor$, $hasSuccessor$, $locatedIn$, $diedIn$, $graduatedFrom$, $hasChild$, $isMarriedTo$
Support	0.05
Possible Positives Covered	200
Positives Covered	50
Speciality ratio	0.1
Learn with free variables	no
Learn with constants	no
Beam search	no
Body length	2

Table 6.1: Parameters for learning the rules

Similarly, we report the average number of facts return for the query $p(?x, ?y)$ as:

$$Average\ \#\ of\ Facts(p) = \frac{\sum_{i=1}^{k_p} \#\ of\ Facts(p,i)}{k_p}$$

where $\#\ of\ Facts(p,i)$ is the number of facts returned for this query in the partition i.

Our experiments show that the rules learned for different partitions are in most cases the same, which means that we can learn efficiently from smaller training sets, after appropriate sampling (see Section 5.2), without missing important rules. In other words, our techniques generalize well across partitions.

In Table 6.1, we can see the parameters that were used while learning the rules in our experiments. As long as the reasoning part is concerned, we should mention that we run URDF with a recursion level of 3, in order to keep the reasoning tractable. Apart from that, the only hard rules that we use in the MAX-SAT solver are those expressing the functional property of $bornIn$ and $diedIn$.

6.3.2 URDF: Experimental Results

In Tables 6.2, 6.3, 6.4 and 6.5, we can see the rules learned for each partition and each confidence threshold with the first argument as input. As we can see the rules are not exactly the same. For example, for partition 2 and 3 for confidence level of 0.90 no rule is learned for the $isMarriedTo$ relation whereas in partition 1 there is a rule learned. This can be due to slight differences in the confidence value of the rule learned in different partitions. So, the rule for the $isMarriedTo$ predicate is a little bit above the 0.9 confidence threshold for partition 1 and a little bit below for partition 2 and 3. In other cases, the different rules learned in each partition is due to slight differences with respect to other thresholds like possible positives to be covered ($E^+(c)$) and positives covered ($n^+(c)$). As we can see from Table 6.5 the confidence level 0.25 is the only level that produces rules for the relations $diedIn$ and $bornIn$, and so only for this level we might have contradictions that need to be resolved with the MAX-SAT solver.

Similarly, the case where the second argument, is depicted in Tables 6.6, 6.7, 6.8 and 6.9. If we compare the rules learned with the first and with the second argument as input, we realize the importance of the choice of the input argument. Not only the confidence values are different (compare confidence values for the $hasSuccessor$ predicate in both cases), but also in some cases we are unable to learn any rules. For example, we are only able to learn rules for the $familyNameOf$ if we use the second argument and for the $livesIn$ if we use the first argument as input. This is due to the different way in which we calculate positives covered ($n^+(c)$) and possible positives to be covered ($E^+(c)$) in both cases. We have already discussed these issues in Section 4.7.

Table 6.10 shows the number of initial facts in the database (Original # of Facts) and the number of facts after the application of rules produced for different confidence thresholds (average among the partitions) when the first argument is the input argument. With * are marked the results in cases, in which we intentionally disabled Max-Sat, and therefore the resulting facts might contain inconsistencies. Table 6.11 shows the precision (average among the partitions) estimated by evaluating manually a sample of 700 facts produced by the rules, for the case in which the first argument is the input. The Wilson interval for a confidence of 95% is illustrated in the brackets. Again we mark with * the results in cases, in which we intentionally disabled Max-Sat. As we can see, as we move to lower confidence levels the precision of the produced facts grows smaller, whereas the amount of facts grows larger. For the predicates $bornIn$ and $diedIn$ which are functional, we can see that the rules produce contradictions which are then resolved by the MAX-SAT solver. Actually, the presence of hard rules for these predicates has as effect the reduction of produced facts (from 36887 to 36443 facts for the $bornIn$ relation and from 22482 to 15240 for the $diedIn$ relation) and an increase to the precision (from 0.78 to 0.91 for the $bornIn$ relation and from 0.22 to 0.29 for the $diedIn$ relation). We can also see how incompleteness can influence the confidence value; the rule for the predicate $isMarriedTo$ in confidence level 0.9 is a rule that expresses the symmetry property of the predicate and therefore the facts produced by this rule have a precision of 1. Nevertheless, the rule has a confidence value of 0.9. But this can be explained if we check how many new facts where produced by the rule and therefore how many facts were missing from the original database. For the predicate $isMarriedTo$, 4207 facts existed initially in the database, while 2361 symmetric facts were missing. A similar situation holds for the predicates $hasSuccessor$ and $hasPredecessor$, for which it holds that the one is the inverse predicate of the other. Note also the difference between the confidence values of the rules learned for the predicate *politicianOf* for the level 0.90 and the measured precision. This difference can have multiple reasons. First of all, the incompleteness of *hasSuccessor* and *hasPredecessor*, which appear in the body of the rules, causes us to measure a higher value for confidence than the actual one. This is obvious if we see how the precision for *politicianOf* drops from level 0.9, where there is no rule learned for the *hasSuccessor* and *hasPredecessor* predicates, and the level 0.85, where there is a rule learned for each of these predicates. The other reason is that the rules for the *politicianOf* predicate are recursive in the sense that the head predicate appears also in the body. In the way we are measuring confidence, we are not taking recursion into consideration. This causes a substantial difference between the confidence value and precision for the recursive rules. The recursion between rules for different predicates can also influence the precision values. For example, false facts produced by the rule:

$$hasChild(x,y)\text{:-}(x \neq y), isMarriedTo(z,x), hasChild(z,y), (z \neq x)[\text{conf: } 0.905]$$

will cause also the production of false facts by the rule:

$$isMarriedTo(x,y)\text{:-}(x \neq y), hasChild(x,z), hasChild(y,z)[\text{conf: } 0.88]$$

This is the reason for the drop in the precision of the *isMarriedTo* predicate between the levels 0.85 and 0.50. This is also very obvious in the case of precision for *politicianOf* in confidence levels 0.85 and 0.50; the rule:

$$locatedIn(x,y)\text{:-}(x \neq y), bornIn(z,x), politicianOf(z,y)[\text{conf:}0.505]$$

produces lots of false facts which are influencing the rule:

$$politicianOf(x;y)\text{:-}bornIn(x,z), locatedIn(z,y), type(x, Governor)[\text{conf:}0.52]$$

In the case of the predicate *graduatedFrom*, the precision is higher than the confidence value of the rules learned. This happens due to incompleteness in the head. In our data, the first argument of graduatedFrom, presents a multiplicity of 1, which implies a functional property of the predicate and makes the computation of confidence very strict, but this does not hold in reality. In Section 4.8, we have already discussed these issues. Apart from that, we can observe that for rules with very low support, like the rules for the predicates *bornIn* and *locatedIn*, the difference between precision and confidence can be large.

Table 6.12 shows the number of initial facts in the database (Original # of facts) and the number of facts after the application of rules produced for different confidence thresholds (average among the partitions) when the second argument is the input argument, whereas Table 6.13 shows the precision (average among the partitions) estimated by evaluating manually a sample of 700 facts produced by the rules and the Wilson interval for confidence 95% (illustrated in the brackets), for the case in which the second argument is the input. Here we can notice the negative influence of the rule

$$hasChild(x, y){:}\text{-}(x \neq y), hasPredecessor(y, x), type(y, Sovereign)$$

on the precision of the predicate $hasChild$ and $isMarriedTo$ (as we have already discussed, those two predicates influence each other). Compare the precision values for these predicates for level 0.50 in Table 6.11, where the above rule is not learned, and in Table 6.13, which includes the influence of this rule. Again, we can observe a large difference between confidence and precision for rules with low support like the rules for the predicate $familyNameOf$.

Partition 1	$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.98 support: 0.37 speciality ratio: 0.28 $politicianOf(x,y)$:-$hasSuccessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.97 support: 0.37 speciality ratio: 0.28 $politicianOf(x,y)$:-$hasPredecessor(z,x), politicianOf(z,y), (z \neq x)$ confidence: 0.97 support: 0.34 speciality ratio: 0.28 $politicianOf(x,y)$:-$hasSuccessor(z,x), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.34 speciality ratio: 0.28 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.90 support: 0.45 speciality ratio: 0.16 $hasChild(x,y)$:-$(x \neq y), isMarriedTo(z,x), hasChild(z,y), (z \neq x)$ confidence: 0.90 support: 0.20 speciality ratio: 0.24
Partition 2	$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.37 speciality ratio: 0.28 $politicianOf(x,y)$:-$hasSuccessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.37 speciality ratio: 0.28 $politicianOf(x,y)$:-$hasPredecessor(z,x), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.35 speciality ratio: 0.29 $politicianOf(x,y)$:-$hasSuccessor(z,x), politicianOf(z,y), (z \neq x)$ confidence: 0.97 support: 0.33 speciality ratio: 0.27
Partition 3	$politicianOf(x,y)$:-$hasPredecessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.37 speciality ratio: 0.28 $politicianOf(x,y)$:-$hasSuccessor(x,z), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.35 speciality ratio: 0.27 $politicianOf(x,y)$:-$hasPredecessor(z,x), politicianOf(z,y), (z \neq x)$ confidence: 0.97 support: 0.33 speciality ratio: 0.27 $politicianOf(x,y)$:-$hasSuccessor(z,x), politicianOf(z,y), (z \neq x)$ confidence: 0.96 support: 0.33 speciality ratio: 0.27

Table 6.2: Rules for confidence level 0.90. Input: 1st argument

Partition 1	$hasPredecessor(x,y)$:-$(x \neq y), hasSuccessor(y,x)$ confidence: 0.87 support: 0.73 speciality ratio: 0.10 $hasSuccessor(x,y)$:-$(x \neq y), hasPredecessor(y,x)$ confidence: 0.87 support: 0.27 speciality ratio: 0.26 $isMarriedTo(x,y)$:-$(x \neq y), hasChild(x,z), hasChild(y,z)$ confidence: 0.88 support: 0.10 speciality ratio: 0.10 $hasChild(x,y)$:-$(x \neq y), isMarriedTo(x,z), hasChild(z,y), (z \neq x)$ confidence: 0.88 support: 0.22 speciality ratio: 0.29
Partition 2	$hasPredecessor(x,y)$:-$(x \neq y), hasSuccessor(y,x)$ confidence: 0.88 support: 0.74 speciality ratio: 0.10 $hasSuccessor(x,y)$:-$(x \neq y), hasPredecessor(y,x)$ confidence: 0.88 support: 0.27 speciality ratio: 0.26 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.87 support: 0.46 speciality ratio: 0.16 $hasChild(x,y)$:-$(x \neq y), isMarriedTo(x,z), hasChild(z,y), (z \neq x)$ confidence: 0.89 support: 0.20 speciality ratio: 0.27 $hasChild(x,y)$:-$(x \neq y), isMarriedTo(z,x), hasChild(z,y), (z \neq x)$ confidence: 0.88 support: 0.18 speciality ratio: 0.24
Partition 3	$hasPredecessor(x,y)$:-$(x \neq y), hasSuccessor(y,x)$ confidence: 0.88 support: 0.74 speciality ratio: 0.10 $hasSuccessor(x,y)$:-$(x \neq y), hasPredecessor(y,x)$ confidence: 0.88 support: 0.26 speciality ratio: 0.26 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.87 support: 0.45 speciality ratio: 0.16 $hasChild(x,y)$:-$(x \neq y), isMarriedTo(x,z), hasChild(z,y), (z \neq x)$ confidence: 0.86 support: 0.18 speciality ratio: 0.23 $hasChild(x,y)$:-$(x \neq y), isMarriedTo(z,x), hasChild(z,y), (z \neq x)$ confidence: 0.89 support: 0.21 speciality ratio: 0.25

Table 6.3: Additional(to level 0.90) rules for confidence level 0.85. Input: 1st argument

Partition 1	$graduatedFrom(x,y)$:-$hasAcademicAdvisor(x,z), worksAt(z,y)$ confidence: 0.51 support: 0.06 speciality ratio: 0.20
Partition 2	$graduatedFrom(x,y)$:-$hasAcademicAdvisor(x,z), worksAt(z,y)$ confidence: 0.51 support: 0.07 speciality ratio: 0.23 $locatedIn(x,y)$:-$(x \neq y), bornIn(z,x), politicianOf(z,y)$ confidence: 0.507 support: 0.012 speciality ratio: 0.14 $isMarriedTo(x,y)$:-$(x \neq y), hasChild(x,z), hasChild(y,z)$ confidence: 0.78 support: 0.12 speciality ratio: 0.13
Partition 3	$politicianOf(x,y)$:-$bornIn(x,z), locatedIn(z,y), type(x, Governor)$ confidence: 0.52 support: 0.09 speciality ratio: 0.26 $locatedIn(x,y)$:-$(x \neq y), bornIn(z,x), politicianOf(z,y)$ confidence: 0.505 support: 0.01 speciality ratio: 0.14 $isMarriedTo(x,y)$:-$(x \neq y), hasChild(x,z), hasChild(y,z)$ confidence: 0.78 support: 0.11 speciality ratio: 0.12

TABLE 6.4: Additional(to level 0.85) rules for confidence level 0.50. Input: 1st argument

Partition 1	$livesIn(x,y)$:-$bornIn(x,y), type(x, Politician)$ confidence: 0.47 support: 0.04 speciality ratio: 0.10 $bornIn(x,y)$:-$livesIn(x,y), type(x, Politician)$ confidence: 0.36 support: 0.02 speciality ratio: 0.26 $diedIn(x,y)$:-$hasPredecessor(x,z), diedIn(z,y), (z \neq x)$ confidence: 0.25 support: 0.12 speciality ratio: 0.21 $graduatedFrom(x,y)$:-$hasAcademicAdvisor(x,z)$, $graduatedFrom(z,y), (z \neq x)$ confidence: 0.32 support: 0.09 speciality ratio: 0.21 $hasPredecessor(x,y)$:-$(x \neq y), hasChild(y,x)$ confidence: 0.40 support: 0.04 speciality ratio: 0.10 $locatedIn(x,y)$:-$(x \neq y), bornIn(z,x), politicianOf(z,y)$ confidence: 0.49 support: 0.01 speciality ratio: 0.12
Partition 2	$livesIn(x,y)$:-$bornIn(x,y), type(x, Politician)$ confidence: 0.36 support: 0.05 speciality ratio: 0.11 $bornIn(x,y)$:-$livesIn(x,y), type(x, Politician)$ confidence: 0.42 support: 0.02 speciality ratio: 0.24 $diedIn(x,y)$:-$hasSuccessor(x,z), diedIn(z,y), (z \neq x)$ confidence: 0.26 support: 0.14 speciality ratio: 0.25 $diedIn(x,y)$:-$hasPredecessor(z,x), diedIn(z,y), (z \neq x)$ confidence: 0.25 support: 0.12 speciality ratio: 0.20 $hasPredecessor(x,y)$:-$(x \neq y), hasChild(y,x)$ confidence: 0.36 support: 0.04 speciality ratio: 0.10
Partition 3	$livesIn(x,y)$:-$bornIn(x,y), type(x, Politician)$ confidence: 0.36 support: 0.05 speciality ratio: 0.11 $bornIn(x,y)$:-$livesIn(x,y), type(x, Politician)$ confidence: 0.41 support: 0.0 speciality ratio: 0.25 $diedIn(x,y)$:-$hasPredecessor(x,z), diedIn(z,y), (z \neq x)$ confidence: 0.26 support: 0.14 speciality ratio: 0.24 $diedIn(x,y)$:-$hasSuccessor(x,z), diedIn(z,y), (z \neq x)$ confidence: 0.25 support: 0.14 speciality ratio: 0.25 $diedIn(x,y)$:-$hasSuccessor(z,x), diedIn(z,y), (z \neq x)$ confidence: 0.26 support: 0.13 speciality ratio: 0.20 $hasPredecessor(x,y)$:-$(x \neq y), hasChild(y,x)$ confidence: 0.35 support: 0.04 speciality ratio: 0.10

TABLE 6.5: Additional(to level 0.50) rules for confidence level 0.25. Input: 1st argument

Partition 1	$hasPredecessor(x,y)$:-$(x \neq y), hasSuccessor(y,x)$ confidence: 0.94 support: 0.73 speciality ratio: 0.10 $hasSuccessor(x,y)$:-$(x \neq y), hasPredecessor(y,x)$ confidence: 0.94 support: 0.26 speciality ratio: 0.24 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.95 support: 0.45 speciality ratio: 0.15
Partition 2	$hasPredecessor(x,y)$:-$(x \neq y), hasSuccessor(y,x)$ confidence: 0.95 support: 0.73 speciality ratio: 0.10 $hasSuccessor(x,y)$:-$(x \neq y), hasPredecessor(y,x)$ confidence: 0.94 support: 0.27 speciality ratio: 0.25 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.92 support: 0.46 speciality ratio: 0.15
Partition 3	$hasPredecessor(x,y)$:-$(x \neq y), hasSuccessor(y,x)$ confidence: 0.94 support: 0.73 speciality ratio: 0.10 $hasSuccessor(x,y)$:-$(x \neq y), hasPredecessor(y,x)$ confidence: 0.94 support: 0.27 speciality ratio: 0.24 $isMarriedTo(x,y)$:-$(x \neq y), isMarriedTo(y,x)$ confidence: 0.93 support: 0.46 speciality ratio: 0.15

Table 6.6: Rules for confidence level 0.90. Input: 2nd argument

Partition 1	$isMarriedTo(x,y)$:-$(x \neq y), hasChild(y,z), hasChild(x,z)$ confidence: 0.88 support: 0.11 speciality ratio: 0.10
Partition 2	$politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasPredecessor(z,x)$ confidence: 0.85 support: 0.37 speciality ratio: 0.27 $politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasSuccessor(z,x)$ confidence: 0.85 support: 0.36 speciality ratio: 0.27 $politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasPredecessor(x,z)$ confidence: 0.86 support: 0.40 speciality ratio: 0.27 $politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasSuccessor(x,z)$ confidence: 0.86 support: 0.39 speciality ratio: 0.27
Partition 3	$isMarriedTo(x,y)$:-$(x \neq y), hasChild(y,z), hasChild(x,z)$ confidence: 0.85 support: 0.11 speciality ratio: 0.11

Table 6.7: Additional(to level 0.90) rules for confidence level 0.85. Input: 2nd argument

Partition 1	$familyNameOf(x,y)$:-$hasChild(y,z), familyNameOf(x,z), (z \neq y)$ confidence: 0.63 support: 0.002 speciality ratio: 0.26 $familyNameOf(x,y)$:-$hasChild(z,y), familyNameOf(x,z), (z \neq y)$ confidence: 0.61 support: 0.002 speciality ratio: 0.27 $hasChild(x,y)$:-$(x \neq y), hasPredecessor(y,x), type(y, Sovereign)$ confidence: 0.58 support: 0.16 speciality ratio: 0.26 $hasChild(x,y)$:-$(x \neq y), hasChild(z,y), (z \neq x), isMarriedTo(z,x)$ confidence: 0.68 support: 0.27 speciality ratio: 0.33 $hasChild(x,y)$:-$(x \neq y), hasChild(z,y), (z \neq x), isMarriedTo(x,z)$ confidence: 0.73 support: 0.25 speciality ratio: 0.33
Partition 2	$familyNameOf(x,y)$:-$hasChild(y,z), familyNameOf(x,z), (z \neq y)$ confidence: 0.57 support: 0.002 speciality ratio: 0.29 $familyNameOf(x,y)$:-$hasChild(z,y), familyNameOf(x,z), (z \neq y)$ confidence: 0.6 support: 0.002 speciality ratio: 0.26 $hasChild(x,y)$:-$(x \neq y), hasPredecessor(y,x), type(y, Sovereign)$ confidence: 0.56 support: 0.14 speciality ratio: 0.23 $hasChild(x,y)$:-$(x \neq y), hasChild(z,y), (z \neq x), isMarriedTo(z,x)$ confidence: 0.62 support: 0.27 speciality ratio: 0.32 $hasChild(x,y)$:-$(x \neq y), hasChild(z,y), (z \neq x), isMarriedTo(x,z)$ confidence: 0.69 support: 0.24 speciality ratio: 0.31 $isMarriedTo(x,y)$:-$(x \neq y), hasChild(y,z), hasChild(x,z)$ confidence: 0.81 support: 0.12 speciality ratio: 0.12
Partition 3	$politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasPredecessor(z,x)$ confidence: 0.80 support: 0.43 speciality ratio: 0.35 $politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasSuccessor(z,x)$ confidence: 0.80 support: 0.43 speciality ratio: 0.35 $politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasPredecessor(x,z)$ confidence: 0.80 support: 0.47 speciality ratio: 0.35 $politicianOf(x,y)$:-$politicianOf(z,y), (z \neq x), hasSuccessor(x,z)$ confidence: 0.80 support: 0.45 speciality ratio: 0.35 $familyNameOf(x,y)$:-$hasChild(y,z), familyNameOf(x,z), (z \neq y)$ confidence: 0.58 support: 0.002 speciality ratio: 0.27 $familyNameOf(x,y)$:-$hasChild(z,y), familyNameOf(x,z), (z \neq y)$ confidence: 0.62 support: 0.003 speciality ratio: 0.27 $hasChild(x,y)$:-$(x \neq y), hasPredecessor(y,x), type(y, Sovereign)$ confidence: 0.58 support: 0.15 speciality ratio: 0.24 $hasChild(x,y)$:-$(x \neq y), hasChild(z,y), (z \neq x), isMarriedTo(z,x)$ confidence: 0.67 support: 0.27 speciality ratio: 0.33 $hasChild(x,y)$:-$(x \neq y), hasChild(z,y), (z \neq x), isMarriedTo(x,z)$ confidence: 0.69 support: 0.26 speciality ratio: 0.34

Table 6.8: Additional(to level 0.85) rules for confidence level 0.50. Input: 2nd argument

Partition 1	
Partition 2	$familyNameOf(x,y)$:-$isMarriedTo(y,z), familyNameOf(x,z), (z \neq y)$ confidence: 0.253 support: 0.006 speciality ratio: 0.31 $familyNameOf(x,y)$:-$isMarriedTo(z,y), familyNameOf(x,z), (z \neq y)$ confidence: 0.25 support: 0.005 speciality ratio: 0.31
Partition 3	$graduatedFrom(x,y)$:-$graduatedFrom(z,y), hasAcademicAdvisor(x,z), (z \neq x)$ confidence: 0.26 support: 0.17 speciality ratio: 0.42

Table 6.9: Additional(to level 0.50) rules for confidence level 0.25. Input: 2nd argument

	# of Facts				
		Confidence Thresholds			
Relation	**Original**	**0.90**	**0.85**	**0.50**	**0.25**
politicianOf	6198	9150	9453	11927	12034
hasSuccessor	55534	-	61681	61681	66058
hasPredecessor	20514	-	61681	61681	66058
isMarriedTo	4207	6568	7025	7940	7940
hasChild	4453	4787	4982	4982	4982
graduatedFrom	4967	-	-	5318	6064
livesIn	14710	-	-	-	16667
bornIn	36188	-	-	-	36887* 36433
diedIn	13618	-	-	-	22482* 15240
locatedIn	60261	-	-	62743	63182

Table 6.10: The increase in the number of facts in the knowledge base. Input: 1st argument.

	Confidence Thresholds			
Relation	**0.90**	**0.85**	**0.50**	**0.25**
politicianOf	0.76 [0.73, 0.79]	0.72 [0.69, 0.75]	0.60 [0.56 0.64]	0.56 [0.53 0.60]
hasSuccessor	-	1 [0.99, 1]	1 [0.99, 1]	0.91 [0.89 0.93]
hasPredecessor	-	1 [0.99, 1]	1 [0.99, 1]	0.88 [0.86 0.90]
isMarriedTo	1 [0.99, 1]	0.96 [0.94, 0.97]	0.88 [0.84, 0.91]	0.88 [0.84, 0.91]
hasChild	0.83 [0.80, 0.86]	0.83 [0.80, 0.85]	0.83 [0.80, 0.85]	0.83 [0.80, 0.85]
graduatedFrom	-	-	0.68 [0.65, 0.72]	0.44[0.41 0.48]
livesIn	-	-	-	0.44[0.406 0.479]
bornIn	-	-	-	0.78[0.75 0.81]* 0.91[0.88 0.92]
diedIn	-	-	-	0.22[0.19 0.25]* 0.29[0.26 0.33]
locatedIn	-	-	0.44 [0.40 0.47]	0.42 [0.38 0.45]

Table 6.11: Precision and confidence intervals. Input: 1st Argument.

	# of Facts				
		Confidence Thresholds			
Relation	**Original**	**0.90**	**0.85**	**0.50**	**0.25**
politicianOf	6198	-	9453	9453	9453
hasSuccessor	55534	61681	61681	61681	61681
hasPredecessor	20514	61681	61681	61681	61681
isMarriedTo	4207	6568	7124	9636	9636
hasChild	4453	-	-	6727	6727
graduatedFrom	4967	-	-	-	6278
familyNameOf	569410	-	-	582085	584505

Table 6.12: The increase in the number of facts in the knowledge base. Input: 2nd argument.

	Confidence Thresholds			
Relation	**0.90**	**0.85**	**0.50**	**0.25**
politicianOf	-	0.725 [0.691, 0.757]	0.725 [0.691, 0.757]	0.725 [0.691, 0.757]
hasSuccessor	1 [0.9945, 1]	1 [0.9945, 1]	1 [0.9945, 1]	1 [0.9945, 1]
hasPredecessor	1 [0.9945, 1]	1 [0.9945, 1]	1 [0.9945, 1]	1 [0.9945, 1]
isMarriedTo	1 [0.9945, 1]	0.98 [0.967, 0.986]	0.707[0.672 0.739]	0.707[0.672 0.739]
hasChild	-	-	0.607[0.57 0.642]	0.607[0.57 0.642]
graduatedFrom	-	-	-	0.214 [0.185 0.246]
familyNameOf	-	-	0.334 [0.30 0.37]	0.34 [0.31 0.38]

Table 6.13: Precision and confidence intervals. Input: 2nd Argument.

6.4 Markov Logic Experiments

6.4.1 Markov Logic: Experimental Setting

The setting for this experiment is in essence a probabilistic reasoning setting. In probabilistic reasoning we present to the user as an answer the inferred facts together with their assigned probabilities. For this experiment, we used the Alchemy tool[1]. Alchemy is able to learn Markov Logic Networks [RD06] in the form of formula-weight pairs, given a database with facts. It can also run inference for some query given the Markov Logic Network. In order to perform inference, Alchemy requires two input files: an evidence (.db) file which contains all the evidence atoms of the query and a Markov Logic Network (.mln) file which contains the formula-weight pairs, a type declaration for each predicate, and a declaration of the constants. Alchemy, then, uses the formula-weight pairs (MLN) and the constants in order to construct a grounded Markov Network. Each predicate will be grounded by using for its arguments any possible constant, which agrees with the argument on the type. The graph of the grounded Markov Network will contain an arc between each pair of atoms that appear together in some grounding of one of the formulas. In the end, Alchemy is able to run inference on the grounded Markov Network according to some user-defined approximation algorithm. The grounding phase in Alchemy can be very expensive and even prohibitive for applications with large databases. If there are n constants and the highest clause arity is c, then the ground network requires $O(n^c)$ memory and the inference time grows super-proportionally. For this reason, we did not use all facts for the construction of input files, but only the facts that were relative to specific predicates, i.e. if for inference for the query $politicianOf(?x, ?y)$ we only used formulas and facts in the URDF-dependency-graph (see Section 3.3) of this query. So, first we run URDF for a query with both arguments being unbound (e.g., $politicianOf(?x, ?y)$) in order to extract a dependency graph, and we use only the facts of the dependency graph while constructing the input files, in order to avoid the quadratic growth in the number of constants of the grounding step in Alchemy. Then we run inference with lifted belief propagation [SD08] on Alchemy for the same query (e.g. $politicianOf(?x, ?y)$) with the rules that we have already learned from our algorithm. Alchemy outputs all new facts (which do not exist in the evidence file) with their probabilities of being true. We then apply filtering according to different probability levels, and we manually measure precision for each level. Precision is measured manually by taking again Wikipedia as a ground truth, and by using a sample instead of all the facts when the amount of produced facts is too large.

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

6.4.2 Markov Logic: Experimental Results

In Table 6.14, the rules that were used in this experiment are listed. Practically, they are the rules learned in the previous experiment (see Section 6.3) for the first argument as input and a confidence level of 0.9. As a first experiment, we used the confidence values of the rules as weights for the formulas. The results are shown in Tables 6.16, 6.17 and 6.18. These tables show the new facts produced by Alchemy together with their precision for each head predicate. The overlap with the facts produced in the experiment in Section 6.3 is also reported. Keep in mind, that the initial facts that exist in the knowledge base for the relations $hasChild$, $politicianOf$ and $isMarriedTo$ are 4453, 6198 and 4207 respectively, and that the precision among only the new facts produced by the same rules in URDF is 0.43, 0.49 and 1, respectively.

As we can see, for the predicate $isMarriedTo$, if we apply a threshold of 0.6, we can separate the facts produced by the rules from the rest of facts that Alchemy produces. Alchemy assigns a probability of around 0.7 to the facts produced by the rules. For the case of the predicate $hasChild$, Alchemy finds it more difficult to separate the facts produced by the rules from the rest of the facts, even when we use high probability thresholds, and the situation becomes even more difficult for the $politicianOf$ predicate. The reason for this is that Alchemy initially assigns a probability of 0.5 to every grounded atom, if not ordered to do otherwise by including a formula-weight pair for a single predicate. In cases, in which the formulas are highly recursive (like it happens for the predicates $hasChild$ and $politicianOf$) this probability gets higher in every iteration of the inference approximation algorithm (in our case lifted belief propagation) and so we end up having also random facts with high probabilities, which are boosted by the cycles in the MLN. In Table 6.15 we can see the formula-weight pairs which correspond to the results of the Tables 6.16, 6.17 and 6.18.

As a second experiment, we tried to use Alchemy to learn the weights of the formulas first and then use the learned weights for the inference step. Unfortunately, the initial network was too large for learning, and the algorithm used for inference afterwards could not converge in a reasonable amount of time. For this reason, we used a smaller network containing only entities representing people for which we also know where they died.

In Table 6.20, we can see the number of facts produced by Alchemy (and their precision), if we learn the weights for the rules. As we can see, for the $isMarriedTo$ relation, all the correct facts produced have a probability close to 0.5, whereas to all incorrect facts probabilities less than 0.1 are assigned. For the relation $politicianOf$ with a threshold of 0.1, we get a precision around 0.92. If we compare the facts produced

politicianOf(x,y):-politicianOf(z,y),hasPredecessor(z,x)
politicianOf(x,y):-politicianOf(z,y),hasSuccessor(z,x)
politicianOf(x,y):-politicianOf(z,y),hasPredecessor(x,z)
politicianOf(x,y):-politicianOf(z,y),hasSuccessor(x,z)

isMarriedTo(x,y):-isMarriedTo(y,x)

hasChild(x,y):-isMarriedTo(x,z), hasChild(z,y)
hasChild(x,y):-isMarriedTo(z,x), hasChild(z,y)

Table 6.14: Rules for the experiment with Alchemy

Inference for Predicate	**Formula-Weight Pairs used**
politicianOf	0.97 politicianOf(x,y):-politicianOf(z,y),hasPredecessor(z,x) 0.96 politicianOf(x,y):-politicianOf(z,y),hasSuccessor(z,x) 0.98 politicianOf(x,y):-politicianOf(z,y),hasPredecessor(x,z) 0.97 politicianOf(x,y):-politicianOf(z,y),hasSuccessor(x,z)
isMarriedTo	0.9 isMarriedTo(x,y):-isMarriedTo(y,x)
hasChild	0.9 isMarriedTo(x,y):-isMarriedTo(y,x) 0.9 hasChild(x,y):-isMarriedTo(x,z), hasChild(z,y) 0.9 hasChild(x,y):-isMarriedTo(z,x), hasChild(z,y)

Table 6.15: Formula-weight pairs. The confidence values are used as weights

Probability Threshold	**New Facts by Alchemy**	**Overlap with URDF**	**Precision**
0.6	1084	659/659=1	0.31 [0.25 0.39]
0.85	615	606/659=0.91	0.4 [0.309 0.49]
0.86	64	55/659=0.08	0.39
0.90	55	55/659=0.08	0.45
0.95	8	8/659=0.01	0.5

Table 6.16: Number of new facts produced by Alchemy for the relation *hasChild* and comparison with the new facts produced by URDF.

Probability Threshold	**New Facts by Alchemy**	**Overlap with URDF**	**Precision**
0.6	2361	2361/2361=1	1
0.7	2361	2361/2361=1	1
0.8	0	0/2361=0	-

Table 6.17: Number of new facts produced by Alchemy for the relation *isMarriedTo* and comparison with the new facts produced by URDF.

Probability Threshold	**New Facts by Alchemy**	**Overlap with URDF**	**Precision**
0.97	3735	212/2952=0.07	0.04 [0.01 0.09]
0.98	2214	106/2952=0.03	0.04 [0.01 0.09]
0.99	383	32/2951=0.01	0.05 [0.02 0.11]

Table 6.18: Number of new facts produced by Alchemy for the relation *politicianOf* and comparison with the new facts produced by URDF.

Inference for Predicate	**Formula-Weight Pairs used**
politicianOf	0.944356 !hasSuccessor(x,y) v politicianOf(x,z) v !politicianOf(y,z) 1.3024 !hasSuccessor(x,y) v politicianOf(y,z) v !politicianOf(x,z) 0.949542 politicianOf(x,y) v !politicianOf(z,y) v !hasPredecessor(x,z) 1.29201 politicianOf(x,y) v !politicianOf(z,y) v !hasPredecessor(z,x) -4.95844 politicianOf(x,y)
isMarriedTo	3.27472 isMarriedTo(x,y) v !isMarriedTo(y,x) -3.09514 isMarriedTo(x,y)
hasChild	2.67182 isMarriedTo(x,y) v !isMarriedTo(y,x) 1.404 !isMarriedTo(x,y) v hasChild(x,z) v !hasChild(y,z) 0.904 !isMarriedTo(x,y) v hasChild(y,z) v !hasChild(x,z) -2.88559 isMarriedTo(x,y) -5.26008 hasChild(x,y)

Table 6.19: Formula - learned weight pairs.

Predicate	**Threshold**	**Facts**	**Precision**	**New Facts**	**Precision for new facts**	**Overlap with URDF**
politicianOf	0.01	193	0.69	141	0.58	99/1199=0.08
	0.1	26	0.92	18	0.88	18/1199=0.01
isMarriedTo	0.1	65	1	65	1	65/65=1
	0.5	65	1	65	1	65/65=1
hasChild	0.01	38	0.36	38	0.36	38/44=0.86
	0.1	1	1	1	1	1/44=0.02

Table 6.20: Precision for each predicate in different probability thresholds for rules with learned weights

by Alchemy and by URDF, we can observe that for the $isMarriedTo$ relation for a threshold of 0.1, Alchemy produces exactly the same facts as URDF. For the $hasChild$ relation and a threshold of 0.01 Alchemy is missing some of the facts produced by URDF, but does not produce random facts. For the relation $politicianOf$ and a threshold of 0.01, Alchemy has only a 8% overlap with URDF and also produces some random facts, but for a threshold of 0.1, the random facts are eliminated. In general, in this setting Alchemy assigns very low probabilities to facts produced by our rules, so we need to use low probability thresholds. This reveals another problem of probabilistic reasoning settings with Markov Logic Networks, namely, the fact that the probability thresholds need to be tuned. Table 6.19 shows the formula-learned weights pairs used for this experiment. The formulas are essentially the same as before, but Alchemy outputs the formulas in disjunctive normal form. Notice that Alchemy learns for each head predicate a formula-weight pair (e.g. -4.95844 $politicianOf(a1, a2)$) with negative weight which makes a randomly grounded atom less probable. Otherwise any grounded atom has around 0.5 probability to be true, as it has already been discussed.

Chapter 7

Summary and Future Work

Information extraction on the Web is inherently incomplete and results in incomplete knowledge bases. One way to tackle with this incompleteness is to use Datalog rules, which hold with some degree of confidence, in order to infer the missing facts. We used the technique of inductive logic programming in order to learn these rules directly from the data for a case of a large and incomplete RDF-knowledge base containing facts extracted from the web.

While trying to apply ILP in such a knowledge base many issues arise. First of all, it is not always obvious how to automatically select positive and negative training examples needed for learning new rules over an incomplete knowledge base and, thus, how to measure the predictive strength of each rule. Apart from that, although powerful, ILP is inherently expensive as there is a combinatorial growth of the search space when constructing these rules, as the size of the background knowledge grows.

In order to overcome these difficulties, we introduced the measure of confidence as a measure for the predictive strength of a rule, which avoids over-penalizing rules because of incompleteness. We used the measures of support and speciality ratio to make sure that the confidence value measured is representative of the reality and that the rule is not overly general, respectively. Apart from that, we investigated the effect of missing facts about known entities on the confidence values and we proposed an alternative confidence definition.

In our implementation for learning rules, we took advantage of the type hierarchy and of the overlap between the entities of different relations in order to restrict the branching factor, one of the main reasons for the high complexity of ILP. Moreover, we investigated sampling for the case of an incomplete knowledge base and in the absence of explicit negative examples. We argued that sampling according to facts can increase the effect of incompleteness and, therefore, that sampling according to entities is preferable. To sum up, we used all the above mentioned measures while implementing an algorithm that allows for early pruning of unpromising branches of the search space, according to a support threshold.

The experimental results show that similar rules were learned from different training data. Apart from that, the confidence value can be indicative of the precision of the rule, if the rule has high support and is not influenced by the presence of other rules.

We believe that there is much work that can still be done on several aspects of applying ILP to large and incomplete knowledge bases:

- **The use of constants:** In our implementation, we only consider the use of constants for the arguments of the head predicate (see Section 5.4.4). The criterion by which we should choose constants for new variables should also be investigated and also to which extent we should allow the usage of constants in rules since their usage enlarges the hypothesis search space and makes the learning process more expensive.
- **Parallelism:** Recently, there has been some discussion about parallelization opportunities in ILP, like in [FSC05], [FSSC09] and [SFJ10]. For applications with large knowledge bases, like the one we are interested in, this research-direction is worth investigating.
- **Taking uncertainty into consideration:** Large knowledge bases extracted from the web exhibit uncertainty, which is expressed by assigning confidence values to facts. So far, we were working with the YAGO [SKW07] dataset, which has an accuracy of 95%, so we were able not to take the uncertainty of the training examples and of the facts in the background knowledge into consideration while learning the rules. One of the most challenging research directions will be to study learning rules from noisy knowledge bases, i.e., when the truth value both of the training examples and of the facts in the background knowledge is uncertain.
- **Sampling:** Work in [MCW$^+$10] implies that sampling can be applied not only to the training data, but also to the background knowledge, which might be very interesting and speed up the learning process a lot for cases in which the background knowledge in very large, like ours.
- **Taking other rules into consideration:** In the experimental part (see Section 6.3), we saw that we have learned the rules one by one, but in reality rules are interdependent. The resulting facts of one rule can cause another rule to fire and thus they can affect its behavior in terms of precision and of the amount of

facts produced. This implies that in the future we should take into consideration in the learning procedure how rules influence one another. This will cause also the confidence value for a rule to be closer to the real precision value, as defined in Section 6.3. The experiments with Alchemy (see Section 6.4) imply that the weights for the formulas should be learned from the data, which also means that the interdependence of the rules should be taken into consideration.

- **Optimize the database part:** In our implementation, the facts are stored in a database and all coverage tests take place in the form of SQL queries to this database. Optimizing the usage of the database, by examining the execution plans for the queries and by investigating a potential benefit by using stored procedures and views or materialized views, might result in a better performance of our system.

Bibliography

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22:207–216, June 1993.

[AvE82] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29:841–862, July 1982.

[CKM+05] John Cabral, Robert C. Kahlert, Cynthia Matuszek, Michael Witbrock, and Brett Summers. Converting semantic meta-knowledge into inductive bias. In *In Proceedings of the 15th International Conference on Inductive Logic Programming*, 2005.

[Cus96] J. Cussens. Part-of-speech disambiguation using ilp. Technical report, Oxford University Computing Laboratory, 1996.

[DJM+98] Sao Deroski, Nico Jacobs, Martin Molina, Carlos Moure, Stephen Muggleton, and Wim Van Laer. Detecting traffic problems with ilp. In David Page, editor, *Inductive Logic Programming*, volume 1446 of *Lecture Notes in Computer Science*, pages 281–290. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0027332.

[DL91] S. Džeroski and N. Lavrač. Learning relations from noisy examples: An empirical comparison of LINUS and FOIL. In L. Birnbaum and G. Collins, editors, *ML91*, pages 399–402. MK, 1991.

[DT98] L. Dehaspe and H. Toivonen. Frequent query discovery: A unifying ilp approach to association rule mining. Technical report, Katholieke Universiteit Leuven, 1998.

[Ď93] S. Džeroski. Handling imperfect data in inductive logic programming. In *Proceedings of the 4th Scandinavian Conference on Artificial Intelligence*, pages 111–125. IOS Press, 1993.

[eK10] Filip elezn and Ondrej Kuelka. Taming the complexity of inductive logic programming. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorn, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 132–140. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11266-9_11.

[FSC05] Nuno A. Fonseca, Fernando Silva, and Rui Camacho. Strategies to parallelize ilp systems. In Stefan Kramer and Bernhard Pfahringer, editors, *Inductive Logic Programming*, volume 3625 of *Lecture Notes in Computer Science*, pages 136–153. Springer Berlin / Heidelberg, 2005. 10.1007/11536314_9.

[FSSC09] Nuno Fonseca, Ashwin Srinivasan, Fernando Silva, and Rui Camacho. Parallel ilp for distributed-memory architectures. *Machine Learning*, 74:257–279, 2009. 10.1007/s10994-008-5094-2.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 48–57, New York, NY, USA, 2003. ACM.

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On the complexity of some inductive logic programming problems. *New Generation Computing*, 17:53–75, 1999. 10.1007/BF03037582.

[HPRW10] Tams Horvth, Gerhard Paass, Frank Reichartz, and Stefan Wrobel. A logic-based approach to relation extraction from texts. In Luc De Raedt, editor, *Inductive Logic Programming*, volume 5989 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13840-9_5.

[KBT$^+$] Jacob Koehler, Jan Baumbach, Jan Taubert, Michael Specht, Andre Skusa, Er Rueegg, Chris Rawlings, and Paul Verrier. S: Graph-based analysis and visualization of experimental results with ondex. *Skusa A, Ruegg A, Rawlings C, Verrier P, Philippi*, 2006.

[KK] Kkt Kakas and Morgan Kaufmann. [ll] lloyd: Foundations of logic programming, springer, 1987. [lst] lloyd, sonenberg, topor: Integrity constraint checking in stratified databases, j. logic programming 4, 1987. [lt] lloyd, topor: Making prolog more expressive, j. logic programming 3, 198.

[KSS95] Ross King, Michael Sternberg, and Ashwin Srinivasan. Relating chemical activity to structure: An examination of ilp successes. *New Generation Computing*, 13:411–433, 1995. 10.1007/BF03037232.

[LA] Jens Lehmann and Sren Auer. Class expression learning for ontology engineering.

[LaB08] Gregor Leban, Jure abkar, and Ivan Bratko. An experiment in robot discovery with ilp. In Filip elezn and Nada Lavrac, editors, *Inductive Logic Programming*, volume 5194 of *Lecture Notes in Computer Science*, pages 77–90. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85928-4_10.

[LCD92] N. Lavrač, B. Cestnik, and S. Džeroski. Use of heuristics in empirical inductive logic programming. In S. Muggleton, editor, *ILP92*, Report ICOT TM -1182, 1992.

[LD93] Nada Lavrac and Saso Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Routledge, New York, NY, 10001, 1993.

[LJFP10] Marco Lippi, Manfred Jaeger, Paolo Frasconi, and Andrea Passerini. Relational information gain. *Machine Learning*, pages 1–21, 2010. 10.1007/s10994-010-5194-7.

[LKR] Niels L, Kristian Kersting, and Luc De Raedt. nfoil: Integrating nave bayes and foil.

[MBR95] Stephen Muggleton, Wolfson Building, and Parks Road. Inverse entailment and progol, 1995.

[MCW$^+$10] Stephen Muggleton, Jianzhong Chen, Hiroaki Watanabe, Stuart Dunbar, Charles Baxter, Richard Currie, Jose Domingo Salazar, Jan Taubert, and Michael Sternberg. Variation of background knowledge in an industrial application of ilp. In *ILP*, 2010.

[MdR94] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

[MF90] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

[Min89] John Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3:319–342, 1989.

[Mit82] Tom M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2):203–226, 1982.

[MS95] Eric Mccreath and Arun Sharma. Extraction of meta-knowledge to restrict the hypothesis space for ilp systems. In *Proceedings of the Eighth Australian Joint Conference on Artificial Intelligence*, pages 75–82. World Scientific, 1995.

[Mug97] Stephen Muggleton. Learning from positive data. In Stephen Muggleton, editor, *Inductive Logic Programming*, volume 1314 of *Lecture Notes in Computer Science*, pages 358–376. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63494-0_65.

[Plo71] G.D. Plotkin. *Automatic Methods of Inductive Inference.* PhD thesis, Edinburgh University, 1971.

[QCj93] J. R. Quinlan and R. M. Cameron-jones. Foil: A midterm report. In *In Proceedings of the European Conference on Machine Learning*, pages 3–20. Springer-Verlag, 1993.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5:239–266, September 1990.

[RD06] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62:107–136, February 2006.

[RFKM08] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008.

[RK08] Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In Raedt et al. [RFKM08], pages 1–27.

[RT10] Luc De Raedt and Ingo Thon. Probabilistic rule learning. In *ILP*, 2010.

[SD08] Parag Singla and Pedro Domingos. Lifted first-order belief propagation. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, pages 1094–1099. AAAI Press, 2008.

[SEWD10] Stefan Schoenmackers, Oren Etzioni, Daniel S. Weld, and Jesse Davis. Learning first-order horn clauses from web text. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 1088–1098, Morristown, NJ, USA, 2010. Association for Computational Linguistics.

[SFJ10] Ashwin Srinivasan, Tanveer Faruquie, and Sachindra Joshi. Exact data parallel computation for very large ilp datasets. In *ILP*, 2010.

[SKW07] Fabian Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A core of semantic knowledge - unifying WordNet and Wikipedia. In Carey L. Williamson, Mary Ellen Zurko, and Prashant J. Patel-Schneider, Peter F. Shenoy, editors, *16th International World Wide Web Conference (WWW 2007)*, pages 697–706, Banff, Canada, 2007. ACM.

[Sri99] Ashwin Srinivasan. A study of two sampling methods for analysing large datasets with ilp, 1999.

[TSSN10] Martin Theobald, Mauro Sozio, Fabian Suchanek, and Ndapandula Nakashole. URDF: Efficient reasoning in uncertain RDF knowledge bases with soft and hard rules. Technical Report MPII20105-002, Max Planck Institute Informatics (MPI-INF), Saarbruecken, Germany, February 2010.

[UM82] Paul E. Utgoff and Tom M. Mitchell. Acquisition of appropriate bias for inductive concept learning. In *AAAI*, pages 414–417, 1982.

[Wil27] Edwin B. Wilson. Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.