TU WIEN Informatics

Weight learning in LP^{MLN} for Collective Classification

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Jacco Oosterhuis
Matrikelnummer 1643191

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Thomas Eiter
Mitwirkung: Tobias Kaminski

Wien, 12. Jänner 2020

Jacco Oosterhuis	Thomas Eiter

TU WIEN Informatics

Weight learning in LP^{MLN} for Collective Classification

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Jacco Oosterhuis
Registration Number 1643191

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Thomas Eiter
Assistance: Tobias Kaminski

Vienna, 12th January, 2020

Jacco Oosterhuis | Thomas Eiter

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jacco Oosterhuis

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Jänner 2020

Jacco Oosterhuis

Acknowledgements

I would like to thank Professor Thomas Eiter and Tobias Kaminski for their continued and extensive patience, support, and help in writing this thesis. Their knowledge on ASP and overall support was invaluable, and writing and finishing this work would not have been possible without their frequent corrections and suggestions over the long period that it took to write this thesis. Thanks also goes to my parents, in supporting and motivating me to finish this thesis and studies.

Abstract

In this thesis we investigate in detail the theory and practice of weight learning in LP^{MLN} for collective classification problems. We test different weight learning methods on a practical collective classification problem, namely object labeling in images using relational context constraints, based on an exposition of the theory of weight learning in LP^{MLN}. In our experiments we consider the applicability of different systems for weight learning in LP^{MLN} and evaluate the performance of different weight learning methods in different scenarios. As we show, the best learning method depends very much on the input program and the specific dataset, and no single method noticeably outperforms other methods in our tests, with very simple learning methods performing very well. The performance of different methods can be very hard to predict, as some of our results are very unexpected. Nonetheless, from our experiments we conclude that weight learning noticeably improves the performance of a LP^{MLN}-program in a collective classification setup and that effective weight learning methods and systems exist for LP^{MLN}.

Contents

CHAPTER 1

Introduction

In many classification and statistical learning tasks, there is a clear logical and relational structure underlying the dataset from which we are trying to learn a model. Taking this structure into consideration can make modeling the data easier and more effective than only considering local features of individual objects. In classification problems, knowing that one object has a certain spatial relation to another object is important information in classifying both objects: if one object is spatially contained in another, the former is unlikely to be an elephant if the latter is a mouse, and vice versa. Therefore, getting the logical and relational structure right is frequently important to learn a model that represents the dataset properly, or can be expected to improve the quality of a model. In standard classification methods, only local or "low-level" features are taken into consideration; standard methods in image classification might consist of functions of pixel intensity or of image gradients, whereas global or relational features might consist of spatial relations between objects.

Recently efforts have been made to take such a relational and logical structure of data into account in statistical learning tasks, in the area of research called *Statistical Relational Learning* (SRL) (Getoor and Taskar 2007). SRL combines a logical or relational understanding of some problem with a probabilistic one, in order to learn statistical models in domains with rich logical structure. The effort to take the local structure of objects into account in classification, so as to jointly determine the correct label assignments of all objects, is called *collective classification* (Sen et al. 2010a). One new formalism in SRL is the LP$^{\text{MLN}}$ formalism (Lee, Meng, and Y. Wang 2015; Lee and Y. Wang 2016a), which extends the semantics of Answer Set Programming (ASP) in a probabilistic way. This is done by extending the stable model semantics of ASP with a probabilistic semantics, which allows to reason purely quantitatively in terms of probabilities over stable models. This allows, for example, LP$^{\text{MLN}}$ to express that worlds where computer mice are found next to keyboards have a higher probability to be true, than worlds where they are not. As such, LP$^{\text{MLN}}$ is able to express that mice are *likely*

to be found near keyboards, without stipulating that this is always the case. It is also a very flexible formalism, which makes it well suited for many SRL tasks; in particular, it is well-suited for collective classification in a object-labeling in images setup, where it has noticeably improved the accuracy of purely local models (Eiter and T. Kaminski 2016).

Central to the probabilistic semantics of LP^{MLN} are the *weights* given to rules in a program, which taken together represent a probability distribution over interpretations (stable models) of an LP^{MLN} program. In order for LP^{MLN} to perform well as a SRL formalism and for prediction tasks, it is of central importance that these weights properly represent the real distribution underlying the dataset. For example, we want the weights to represent the real probability that a weight is found next to a keyboard, and to give a higher probability of being close to a keyboard than being close to a refrigerator, say. Barring the possibility of hand-coding the weights by using expert knowledge — something that gets increasingly more complex as the size of the LP^{MLN} program increases, and can take multiple weeks to complete — it is desirable to be able to learn the weights from a set of examples generated from a distribution we wish to model.

So far, weight learning in LP^{MLN} has not been extensively studied. Recently, in (Lee and Y. Wang 2018), a native formalism was proposed in the form of the LP^{MLN}-learn system (*LPMLN learning system* 2019), and translations to MLN-learning systems exist due to (Lee, Talsania, and Y. Wang 2017b; Lee, Talsania, and Y. Wang 2019), but to our knowledge no detailed account and overview of both the theory and practice of weight learning in LP^{MLN} exists. What is required is an exposition of the theoretical aspects of weight learning in LP^{MLN}, systems available for supervised weight learning which can be used on LP^{MLN} programs, and practical tests on efficacy of methods and systems in different weight learning scenarios. Furthermore, practical tests can give insight into practical issues encountered in weight learning in LP^{MLN}. Specifically for SRL, no investigation has yet been performed into the accuracy of models learned using different methods, although accuracy is one of the most informative measures on the quality of a weight learning method in prediction tasks.

In this thesis we intend to close this gap and investigate supervised weight learning for LP^{MLN} in a collective classification setting. The work done here is meant to both give an overview of the theory of weight learning in the LP^{MLN} formalism, as well as investigate weight learning on a real-life application. In this work we attempt to answer the following questions:

- What algorithms can be used for weight learning in LP^{MLN}, and how do algorithms perform under specific conditions (e.g. missing data)?
- What systems exist for weight learning in LP^{MLN} and how do they perform and differ in terms of speed and accuracy?
- What are practical difficulties encountered when learning weights of an LP^{MLN} program?

- How does the structure of an LP^{MLN} program affect the effectiveness of weight learning methods?

As a test application we will take the object-labeling in images setup of (Eiter and T. Kaminski 2016), and compare different methods for weight learning on the two datasets used in this classification problem. This classification setup is a collective classification problem, where the aim is to take spatial relations between objects into account in assigning labels to objects. We will investigate and test existing systems which can be used for weight learning, to test whether certain implementations perform better than others on our program. To investigate the performance of methods in different scenarios, we will also perform different experiments; most notably learning of the weight of each constraint in the LP^{MLN} program independently from the others, and learning on missing data.

From our experiments we conclude that the most effective learning method depends on the LP^{MLN} program whose weights are to be learned, as we find that a very simple log-odds calculation performs very well on our datasets and input programs. In our experiments, we did generally notice a strong improvement in the quality of the LP^{MLN} program when learning the weights, compared to a uniform weight attribution to the rules, although some learning methods also failed to produce a meaningful model in some of our experiments. In some tests, such as learning on missing data, we surprisingly found the learning methods to be very stable. We also encountered problems with learning in some systems, which made the number of systems available smaller than anticipated. Overall, the effectiveness of specific methods for learning is hard to predict, and seems to depend very much on the specific LP^{MLN} program and dataset.

This work is structured as follows. In Chapter 2 we will introduce the LP^{MLN} formalism and compare this to related formalisms. In Chapter 3 the theory behind weight learning for LP^{MLN} is studied, where we will compare and investigate different algorithms for weight learning, along with different scenarios which we expect to have an impact on the efficacy of some methods. This chapter will also motivate some hypotheses about weight learning. In Chapter 4 we perform experiments on a classification setup using LP^{MLN}, based around hypotheses motivated by Chapter 3. Lastly, in Chapter 5 we will summarize our findings and discuss further avenues of research related to this work.

CHAPTER 2

LP$^{\text{MLN}}$

2.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative problem-solving approach, based on the *stable model* or *answer set* semantics. The ASP formalism is a powerful reasoning and knowledge representation framework with large expressive power: it extends first-order logic in multiple ways, most notably the power of expressing incomplete information and the possibility of expressing transitivity, but borrows the clearness and conciseness of first-order logic.[1]

Both the syntax and semantics of ASP are similar to those of standard first-order logic, but involves noticeable differences. In this work we will, following (Lee and Y. Wang 2016a), restrict ourselves to a first-order signature σ that contains no function symbols of positive arity, and follow the semantics as described in (Gelfond and Lifschitz 1988). As is standard, a first-order interpretation consists of a domain D, a mapping from constant symbols in σ to elements of D, a mapping from relation symbols to tuples over D, and a mapping from function symbols to functions (of the same arity) over D.

In the ASP formalism a *disjunctive normal logic program* Π is a finite set of rules of the form

$$Head \leftarrow B_1, ..., B_k, not\ B_{k+1}, ..., not\ B_n \tag{2.1}$$

where $Head$ is a disjunction of first-order atoms, all B_i are first-order atoms, and *not* is default negation corresponding to negation as failure. In a rule of a program Π, the $B_1, ..., B_k$ make up the body, $Head$ is called the "head", and $B_{k+1}, ..., B_n$ make up the *negative body*. We will occasionally in this work refer to the combined body with $Body$. As is usual in the literature, the comma is used for conjunction, and semicolon for disjunction (in $Head$). If a logic program contains no variables, it is *ground*; a *grounding*

[1] See (Brewka, Eiter, and Truszczyński 2011) for a good introduction to ASP and logic programming.

of a logic program Π with respect to a signature σ, denoted $gr_\sigma(\Pi)$, corresponds to replacing every variable with every ground term of σ. If $Head$ is empty in a rule R (or, equivalently, $\bot$), R is called a *constraint*. If the body of a rule equals *true* (or $\top$) (and $Head$ is not empty), R is called a *fact*.

Disjunctive normal logic programs can be extended by adding strict negation $\neg$ in addition to the default negation *not*; such programs are called *extended logic programs*. As such, in extended logic programs the syntax uses first-order literals possibly preceded by *not*, whereas normal logic programs use first-order atoms possibly preceded by *not*. The stable models are called *answer sets* in this case. However by introducing positive predicates P' (fresh predicates) for every negative literal P and leaving the rest as is, any extended logic program Π can be translated to an equivalent normal logic program Π^+ obtained by substituting any negative literal by fresh positive literals. Denoting $\mathcal{I}^+$ as the interpretation obtained by replacing negative literals as just described, any consistent interpretation $\mathcal{I}$ is an answer set if and only if $\mathcal{I}^+$ is a stable model to the corresponding positive normal logic program (Gelfond and Lifschitz 1991). As such, we will frequently restrict ourselves without loss of generality to normal logic programs in this work.

As an example of a normal logic program, social network influences can be easily modeled as done in the following program:

Program 2.1

$$Friends(a, b)$$
$$Friends(b, c)$$
$$Influences(x, y) \leftarrow Friends(x, y), not\, distant\_Friends(x, y)$$
$$Influences(x, z) \leftarrow Influences(x, y), Influences(y, z)$$

This states that the $Influences$ relation is transitive. The third rule, with default negation, here expresses that if there is no information (it cannot be proven) that, for any two friends x, y, they are distant friends, then they influence each other. The *not* here allows the expression of incomplete information, and is different from the explicit negative information contained in classical negation $\neg$. As is well known[2], transitive closure cannot be expressed within first-order logic, but can be easily expressed in ASP. If we consider a social network as a graph, with persons as nodes in the graph and edges between two nodes if and only if they are friends, and express reachability in that graph using the predicate $Reach$, then the following example easily expresses the transitive closure of the $Friends$ relation:

Program 2.2

$$Reach(x, x) \leftarrow Person(x)$$
$$Reach(x, z) \leftarrow Friends(x, y), Reach(y, z) \quad .$$

[2]For the case of infinite structures, this follows from the compactness theorem. The result also holds for finite structures, which can be proven using the theorem of Ehrenfeucht-Fraïssé.

This defines the transitive closure of the $Friends$ relation: pairs of people x, y such that y is reachable in the graph from x using the Friends relation by any non-negative number of steps (that is, following zero or more edges of nodes that are in the $Friends$ relation).

We can rewrite a rule R as a first-order formula F_R by replacing *not* with standard negation $\neg$, replacing commas by conjunction $\wedge$, semicolons by disjunction $\vee$, and treating $Head \leftarrow Body$ as implication in the other direction, $Body \rightarrow Head$. Constraints, $\leftarrow Body$ should be translated into $\neg(Body)$. Any variables in R should be universally quantified. For example, the second rule of the above example corresponds to the formula:

$$\forall x \forall y \forall z((Friends(x, y) \wedge Reach(y, z)) \rightarrow Reach(x, z))$$

A ground program Π is then identified with the conjunction of the formulas corresponding to all rules $R \in \Pi$, denoted F_Π.

The *Gelfond-Lifschitz Reduct* (or simply *reduct*) of Π relative to an interpretation $\mathcal{I}$, denoted $\Pi^\mathcal{I}$, corresponds to the program obtained by deleting:

1. Every rule R that has a negative literal *not* B in its body, such that $B \in \mathcal{I}$;
2. All negative bodies of remaining rules in Π.

The semantics of ASP programs can be given in terms of Herbrand interpretations: interpretations where every constant is interpreted as itself, and every function symbol is interpreted as the corresponding function. If we restrict σ to be finite, this yields only finitely many Herbrand interpretations. A Herbrand interpretation $\mathcal{I}$ is a *stable model* of a ground program Π if $\mathcal{I} \models \Pi^\mathcal{I}$, and if $\mathcal{I}$ is minimal with respect to set inclusion. Here $\mathcal{I} \models \Pi^\mathcal{I}$ is short for $\mathcal{I} \models F_{\Pi^\mathcal{I}}$ as defined above. $\mathcal{I}$ is thus a stable model of Π if it is a minimal Herbrand model of the reduct $\Pi^\mathcal{I}$. For a non-ground program Π, $\mathcal{I}$ can be defined to be a stable model of Π if and only if it is a stable model of the grounding $gr_\sigma(\Pi)$ of Π.[3]

To illustrate this semantics with respect to our social network example 2.1, over the Herbrand Universe $\{a, b\}$, we obtain four possible groundings of the third rule, by replacing x, y by the four possible combinations of a, b. The third rule then corresponds to the first-order formula:

$$(Friends(x, y) \wedge \neg distant\_Friends(x, y)) \rightarrow Influences(x, y).$$

Corresponding to the *minimal model* or *stable model* semantics, we look for the minimal model that satisfies the groundings of this formula: this will be a minimal Herbrand interpretation $\mathcal{I}$ where, for any constants a, b in the language, if $Friends(a, b) \in \mathcal{I}$ and

[3]The stable model semantics has also been defined for first-order sentences, see (Ferraris, Lee, and Lifschitz 2011). This semantics in terms of first-order sentences can prevent us from having to first construct all the groundings of a formula or rule, which might make it computationally much more efficient. See also Section 3.3.4.

$distant\_Friends(a,b) \notin \mathcal{I}$ (corresponding to the reduct), then $Influences(a,b) \in \mathcal{I}$; this is different from an interpretation that satisfies $\neg distant\_Friends(a,b)$.

For practical usage and inference using ASP solvers, it is important that any logic program has only finitely many ground instances, such that for any logic program, an equivalent ground logic program can be computed. For this the requirement of a variable being *safe* is introduced (Eiter, Mehuljic, et al. 2015, p. 40)[4]:

Definition 1. Safety
A variable x occurring in a rule R is called *safe* if x occurs in an atomic formula or strongly negated standard literal (not involving default negation) in R. A rule is called safe if all variables occurring in the rule are safe.

For instance, the following program is not safe

Program 2.3

$$Friend(x,y) \leftarrow not\ Strangers(x,y)$$

Whereas the following program is:

Program 2.4

$$Friend(x,y) \leftarrow not\ Strangers(x,y), Person(x), Person(y)$$

Extensions within ASP

The formalism as described can be extended in numerous ways. An extension of ASP is to extend the syntax with double default negation, *not not A*. This is allowed in the input language of *gringo*(Gebser et al. 2017). Terms such as *not not A* are satisfied whenever their positive counterparts are; absorption, as in double classical negation, does not hold, however. To see this, consider the two following programs:

Program 2.5

$$B \leftarrow A$$
$$A \leftarrow B$$

which only has the empty set $\emptyset$ as a stable model, and

Program 2.6

$$B \leftarrow not\ not\ A$$
$$A \leftarrow not\ not\ B$$

which has the empty set $\emptyset$, but also $\{A, B\}$ as a stable model. This is because both A and B under double default negation do not need acyclic derivations, unlike in the absorbed case. Here *not not A is* equivalent to $not\ \neg A$.

[4]The definition given here does not cover arithmetic predicates and aggregates.

Further constructs are "choice rules" $\{A\}^{ch} \leftarrow B$, which are shorthand for two rules

Program 2.7

$$A \leftarrow B, not\ \overline{A}$$
$$\overline{A} \leftarrow B, not\ A$$

where $\overline{A}$ represents the classical negation of A (as described above), and neither A nor $\overline{A}$ occur in the head of any other rule in Π. This choice construct has the effect that exactly one of A and $\overline{A}$ is included in any answer set, if B is satisfied.

In addition more constructs are possible — e.g. counting constructs and aggregates — but these will not be discussed further in this work; the probabilistic formalism LP$^{\text{MLN}}$ introduced later in this chapter can be extended to incorporate these as well (Lee, Meng, and Y. Wang 2015, p. 3).

Weak Constraints

As with traditional logic, ASP as presented above suffers from the limitation that the knowledge represented in the programs is strict: it does not allow to incorporate degrees of belief or considerations of likelihood over propositions. The extension to ASP of *weak constraints* tries to incorporate degrees of belief in the semantics of ASP. Weak constraints are rules (more precisely, constraints) of the form:

$$:\sim B_1, ..., B_k, not\ B_{k+1}, ..., not\ B_n[w@l]$$

where w, a positive integer, is the *weight* of the constraint, and l is a non-negative integer, which stands for the l-th *level* of the constraint (Calimeri et al. 2012). The levels encode which weak constraints are to be minimized (with respect to w) first in computing the stable models: the idea is that weak constraints are optimized in a stepwise manner, starting at the highest level.[5]

The stable models of a program with weak constraints $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 does not contain weak constraints and Π_2 is a set of weak constraints, are defined as the stable models of Π_1. In addition, we seek stable models that are optimal; i.e., those stable models with minimum *penalty*. The penalty $Pen_\Pi(\mathcal{I}, l)$ at the l-th level of an interpretation $\mathcal{I}$ is given as:

$$Pen_\Pi(\mathcal{I}, l) = \sum_{\substack{:\sim Body\,[w@l] \in \Pi_2, \\ \mathcal{I} \models Body}} w,$$

where $Body\,[w@l]$ is used for the entire positive and negative body of the weak constraint, with weight w at level l.

These weak constraints therefore encode rules whose body it is desirable not to satisfy. We then say that an interpretation $\mathcal{I}$ is *dominated* by another $\mathcal{I}'$ if:

[5] As shown in (Buccafurri, Leone, and Rullo 2000), the levels can be incorporated into the weights. However, because the normal translation of LP$^{\text{MLN}}$ to ASP with weak constraints uses the levels, it is useful to keep these in the formalism.

- there is some nonnegative integer l such that $Pen_{\Pi}(\mathcal{I}', l) < Pen_{\Pi}(\mathcal{I}, l)$ and
- for all $k > l$, $Pen_{\Pi}(\mathcal{I}', k) = Pen_{\Pi}(\mathcal{I}, k)$.

A stable model is then said to be optimal if it is not dominated. In solving ASP with weak constraints, we seek for optimal stable models, intuitively meaning we minimize the penalty of the weak constraints.

Here it is important to note that neither the weights nor penalties of the weak constraints do not correspond to probabilities. The benefit of the formalism is that it allows to encode degrees of uncertainty in logic programming.

2.2 Markov Logic

An influential attempt to introduce a probabilistic semantics for first-order logic is Markov Logic (Richardson and Domingos 2006). In first-order logic, if an interpretation violates some formula in a knowledge base (a set of formulas that hold in some dataset), that interpretation is impossible. This is parallel to ASP, where an interpretation that does not satisfy some atom of a knowledge base cannot be an answer set. As we have seen, weak constraints do allow for some leeway here, but the weights cannot be interpreted as probabilities: the axioms of probability are not guaranteed to be satisfied by assignments of weights to weak constraints.

Markov Logic builds upon classical first-order logic in a probabilistic way by assigning weights to first-order formulas. Under specific conditions, these weights then define a probability distribution over interpretations satisfying the Markov Logic Network (MLN).

Definition 2. Markov Logic Network
A Markov Logic Network (MLN) is a set of pairs (F_i, w_i), where F_i is a first-order formula, and w_i is a real-valued weight.

As a simple example, one can have the following MLN:

$$\forall x \forall y Friends(x, y) \rightarrow Influences(x, y) \qquad 2.5$$
$$\forall x \forall y (Smokes(x) \wedge Influences(x, y)) \rightarrow Smokes(y) \qquad 1.5$$

This MLN encodes that friends are quite likely to influence one another, and that smokers might influence other people to smoke, too. This second statement is however encoded as less likely or less important—it has lower weight—than the sentence that says that friends influence one another. The weight of a formula F can here be interpreted as the log odds between a world where F is true and one where F is false (Richardson and Domingos 2006, p. 115). Alternatively, these weights can be seen as penalties on interpretations for failure to satisfy a formula.

2.2.1 Markov Networks

Markov Logic differs from the approach of weak constraints in ASP by specifying weights for all formulas, and in ascribing a probabilistic semantics to these weighted formulas. It derives this probabilistic semantics from *Markov Networks*. A Markov Network, also known as a *Markov Random Field*, is a graphical probabilistic model for a joint distribution of a set of random variables $\mathbf{X} = \{X_1, X_2, ..., X_n\}$, where every random variable X_i in this set is a function over some sample space or domain S_i. Markov Networks can conveniently be represented as undirected graphs $G = (V, E)$, where the random variables are nodes V, and the edges E in the graph are such that every random variable X_i is independent of other random variables, given its neighbors $\mathcal{N}_G(X_i)$ in the graph G: $P(X_i|\mathbf{X}\backslash\{X_i\}) = P(X_i|\mathcal{N}_G(X_i))$. The power of Markov Networks in probabilistic modeling lies in that it can represent a joint distribution compactly, and allows to factorize the joint distribution in an efficient though general way.

The reason for this is that there is a strong relation between probability distributions that satisfy the local independence property $P(X_i|\mathbf{X}\backslash\{X_i\}) = P(X_i|\mathcal{N}_G(X_i))$ for a graph G, and probability distributions that *factorize* over a Markov Network M. A factor ϕ is simply a real-valued function over the possible values of a subset of $\mathbf{X}$. We say that a probability distribution P factorizes over a Markov Network M, if the distribution factorizes into factors over complete subgraphs of M:

$$P(\mathbf{X}) = \frac{1}{Z} \prod_{j=1}^{k} \phi_j(D_j[\boldsymbol{x}]), \tag{2.2}$$

for a set of factor functions $\Phi = \{\phi_1, ..., \phi_k\}$ over k complete subgraphs $D_1, ..., D_k$ of M, and a normalizing constant function Z that corresponds to summing over all n possible values of $\mathbf{X}$:

$$Z = \sum_{X_1,..,X_n} \prod_{m} \phi_m(D_m).$$

The $D_j[\boldsymbol{x}]$ stands for the state of the j-th subgraph, given $\boldsymbol{X} = \boldsymbol{x}$.[6]

As a clique is a maximal connected subgraph, it is sufficient if $P_{\mathcal{H}}$ factorizes over the cliques of $\mathcal{H}$, but other (more efficient) completely connected subgraphs can also be used. In the case of the factors ϕ being functions of cliques, the factors are also called clique potentials.

It is important to note that factors are generally not (marginal) probabilities, and are only an unnormalized local measure of odds; the behavior of a factor can be completely in contrast with the overall (combined) joint distribution. This is in contrast to Bayesian networks where factors correspond to marginal probabilities, and this also makes inference and learning of Markov Networks a hard problem: the total joint distribution must be calculated, which involves calculating the partition function Z. This fact will be important later, when we consider learning in LP^{MLN}.

[6] For a proof of the result and elaborate explanation, see (Koller and Friedman 2009, pp. 114–121).

2.2.2 Markov Logic Networks

A Markov Logic program or Markov Logic Network $\mathbb{L}$ can be seen as a *template* for a class of Markov Networks. It defines a probability distribution over Herbrand Interpretations, also called *possible worlds* in this context (Richardson and Domingos 2006), by constructing, given a finite set of constant symbols C, a unique corresponding Markov Network $M_{\mathbb{L},C}$:

- For every possible grounding of every predicate P in $\mathbb{L}$, there is a binary node in $M_{\mathbb{L},C}$, whose value is 1 if the ground atom is true, and 0 otherwise.
- For every possible grounding of every formula F in $\mathbb{L}$, there is one feature function f_i in $M_{\mathbb{L},C}$, whose value is 1 if the ground formula is true, and 0 otherwise. The weight w_i in $M_{\mathbb{L},C}$ is the weight of the formula in the Markov Logic Network.

Such a Herbrand interpretation or possible world assigns a truth value to each possible ground atom. Here grounding an MLN is exactly the same as in first-order logic or as in ASP, as explained earlier. The probability distribution has the same form as that in Markov Networks, but is defined here over the set of Herbrand interpretations (finite because of the restriction on σ), denoted $\mathfrak{I}$. The probability of a Herbrand interpretation $\mathcal{I} \in \mathfrak{I}$ is then given by:

$$P_{M_{\mathbb{L},C}}(\mathcal{I}) = \frac{1}{Z} \exp\left[\sum_{i=1}^{m} w_i n_i(\mathcal{I})\right] = \frac{1}{Z} \prod_{i=1}^{m} \phi_i(D_i[\mathcal{I}])^{n_i(\mathcal{I})}, \tag{2.3}$$

where:

- $n_i(\mathcal{I})$ is the number of times the formula F_i is true in $\mathcal{I}$, or equivalently the number of true groundings of F_i in $\mathcal{I}$;
- $D_i[\mathcal{I}]$ is the state of the atoms in F_i, given $\mathcal{I}$;
- Z is the partition function, given by $\sum_{\mathcal{J}\in\mathfrak{I}} \exp\left[\sum_{i=1}^{n} w_i n_i(\mathcal{J})\right]$, respectively $\sum_{\mathcal{J}\in\mathfrak{I}} \prod_{i=1}^{n} \phi_i(D_i[\mathcal{J}])^{n_i(\mathcal{J})}$;
- m is the number of ground formulas.

Any interpretation $\mathcal{I}$ is called a *model* if $P_{M_{\mathbb{L},C}}(\mathcal{I}) \neq 0$.

Given the construction of $M_{\mathbb{L},C}$ from a MLN $\mathbb{L}$ and a set of constants C, the graphical structure of $M_{\mathbb{L},C}$ is such that there is an edge between two nodes if and only if the corresponding ground atoms appear together in at least one grounding of a formula in $\mathbb{L}$ (Richardson and Domingos 2006, p. 112). These atoms in a ground formula therefore form a clique (completely connected subgraph); the ϕ_i are therefore frequently called clique potentials. In the case of MLN, $\phi_i(D_i[\mathcal{I}])$ is given by $\exp[w_i]$.

The partition function Z normalizes the exponential weighted sum so that $P_{M_{\mathbb{L},C}}$ is really a probability distribution, by summing over the other possible Herbrand interpretations or possible worlds. To this end, $P_{M_{\mathbb{L},C}}$ can be written in a slightly more straightforward form by decoupling the unnormalized function from the partition function. For this, first define $\mathbb{L}_{\mathcal{I}}$ as the set of ground formulas in a ground MLN that are satisfied by $\mathcal{I}$. Then we can define the *unnormalized weight* $W_{M_{\mathbb{L},C}}(\mathcal{I})$ as:

$$W_{M_{\mathbb{L},C}}(\mathcal{I}) = \exp\left[\sum_{\substack{w:F\in\mathbb{L}\\ F\in\mathbb{L}_{\mathcal{I}}}} w\right].$$

The probability $P_{M_{\mathbb{L},C}}(\mathcal{I})$ is then given by:

$$P_{M_{\mathbb{L},C}}(\mathcal{I}) = \frac{W_{\mathbb{L}}(\mathcal{I})}{\sum_{\mathcal{J}\in\Im} W_{\mathbb{L}}(\mathcal{J})}.$$

This follows immediately from the definition of $n_i(\mathcal{I})$ as the number of true groundings of F_i, given $\mathcal{I}$. As we shall see in the next chapter, the partition function complicates inference and learning in MLN (and Markov Networks) quite a bit.

Any first-order theory, or *knowledge base*, can be easily encoded as a MLN, by setting the weights to arbitrarily high values. As w goes to infinity, the probability given by the corresponding MLN represents a uniform distribution over the Herbrand interpretations that satisfy the knowledge base. Because the probability of a formula in MLN is given by the sum over the interpretations that satisfy the formula, any formula that is true in the knowledge base obtains probability 1 as w goes to infinity.

Proposition 1. *(Richardson and Domingos 2006, p. 115)*
Let KB be a satisfiable first-order knowledge base, $\mathbb{L}$ be the MLN obtained by assigning weight w to every formula in KB, C be the constants appearing in KB, $\Im_{KB}$ be the set of interpretations $\mathcal{I} \in \Im$ that satisfy KB, and F be an arbitrary first-order formula. Then:

1. $\forall \mathcal{I} \in \Im_{KB} \lim_{w\to\inf} P(\mathcal{I}) = |\Im_{KB}|^{-1}$
   $\forall \mathcal{I} \notin \Im_{KB} \lim_{w\to\inf} P(\mathcal{I}) = 0$
2. $\forall F, KB \models F$ *if and only if* $\lim_{w\to\inf} P(F) = 1$.

Proof.

- If $\mathcal{I} \in \Im_{KB}$ then $P(\mathcal{I}) = \frac{\exp(kw)}{Z}$.
  If $\mathcal{I} \notin \Im_{KB}$ then at least one formula in KB is not satisfied under $\mathcal{I}$, so $P(\mathcal{I}) \leq \frac{\exp((k-1)w)}{Z}$. Therefore all $\mathcal{I} \in \Im_{KB}$ are equiprobable, with $P(\Im_{KB}) \leq |\Im_{KB}|^{-1}$.
  From this we have, as $\lim_{w\to\inf}$:

$$\frac{P(\Im \setminus \Im_{KB})}{P(\Im_{KB})} \leq \frac{\sum_{\mathcal{I}\notin\Im_{KB}} P(\mathcal{I})}{\sum_{\mathcal{I}\in\Im_{KB}} P(\mathcal{I})} \leq \frac{(\exp\left[(k-1)*w\right]/Z) * |\Im \setminus \Im_{KB}|}{(\exp\left[k*w\right]/Z) * |\Im_{KB}|} = 0$$

where the second inequality comes from the fact that every $\mathcal{I} \notin \Im_{KB}$ does not satisfy at least one formula in KB (of weight w). This proves 1.

- Let $\Im_F$ denote the set of interpretations that satisfy F, for any F. If $KB \models F$, then $\Im_{KB} \models F$ and so $P(F) = \sum_{\mathcal{I} \in \Im_F} P(\mathcal{I}) \geq P(\Im_F) \geq P(\Im_{KB})$; so by part 1 if $KB \models F$ then $\lim_{w \to \inf} P(F) = 1$. For the converse, if $\lim_{w \to \inf} P(F) = 1$, then $\forall \mathcal{I}((P\mathcal{I}) > 0 \to \mathcal{I} \models F)$. Since we have from part 1, that $\forall \mathcal{I} \notin \Im_{KB} \lim_{w \to \inf} P(\mathcal{I}) = 0$, $\Im_{KB} \subseteq \Im_F$, so we have $KB \models F$.

□

Even if the weights are set to finite values, for any set of constants C, the satisfying assignments of formulas in a KB are the maximum values (the modes) of the distribution of $M_{\mathbb{L},C}$ (as given by Eq. (2.3)). This can be seen by taking the logarithm:

$$\ln(P_{M_{\mathbb{L},C}}(\mathcal{I})) = \sum_{i=1}^{m} w_i n_i(\mathcal{I}) - \ln(Z).$$

This value is maximal for those $\mathcal{I}$ with highest n_i (number of times F_i is true in $\mathcal{I}$) for every $F \in KB$, which are those $\mathcal{I}$ s.t. $\mathcal{I} \models KB$.

To illustrate how MLN extends first-order logic, consider the simple MLN consisting of only the formula $\forall x \forall y Smokes(x) \to hasCancer(x))$ with arbitrary weight w, with $C = \{Alice\}$. This leads to the following possible worlds or Herbrand interpretations:

$$\{\neg Smokes(Alice), \neg hasCancer(Alice)\}$$
$$\{Smokes(Alice), hasCancer(Alice)\}$$
$$\{Smokes(Alice), \neg hasCancer(Alice)\}$$
$$\{\neg Smokes(Alice), hasCancer(Alice)\}$$

As given by Eq. (2.3), the first, second, and fourth interpretation all have probability $\frac{e^w}{3e^w+1}$, while the third has probability $\frac{1}{3e^w+1}$. Depending on the choice of w the third interpretation is not necessarily impossible, although generally significantly less likely than the other interpretations.

2.3 LP$^{\text{MLN}}$

Markov Logic extends first-order logic in a probabilistic way, but it lacks some of the expressive power and ease of representation of ASP. This is because it inherits the semantics of first-order logic, and can thus not express transitive closure correctly.

A probabilistic semantics has been constructed for logic programs in the LP$^{\text{MLN}}$ formalism, which builds upon MLN by using the logic programming formalism, and extends ASP by using the MLN semantics. LP$^{\text{MLN}}$ is a relatively new extension of Answer Set

Programming proposed in (Lee and Y. Wang 2016a), where all rules of a program are given weights. Formally, an LP$^{\text{MLN}}$ program is a finite set of weighted rules $R : w$, where R is a rule of a disjunctive normal logic program as described in the previous section, and w is either a real number or α, denoting "infinite weight". In the case that w is a real number, $R : w$ is called a *soft rule*; in the case of α a *hard rule*. As in the case of ASP, a program without variables is called ground; here any grounding of a non-ground rule receives the same weight as the corresponding non-ground rule. Also, an LP$^{\text{MLN}}$ program Π is called safe if and only if the unweighted program $\overline{\Pi}$ is safe.

Following notation of Lee and Y. Wang, we will use $\overline{\Pi}$ to denote the set $\{R \mid w : R \in \Pi\}$, which corresponds to disregarding the weights of an LP$^{\text{MLN}}$ program. To denote the subset $w : R$ of Π such that $\mathcal{I} \models R$ we use $\Pi_{\mathcal{I}}$, and $SM[\Pi]$ denotes the set of stable models of subsets of $\overline{\Pi}$: $SM[\Pi] := \{\mathcal{I} \mid \mathcal{I} \text{ is a stable model of } \overline{\Pi_{\mathcal{I}}}\}$. Here it is important that the models in $SM[\Pi]$ need *not* satisfy all the rules in the program: every element $\mathcal{I}$ of $SM[\Pi]$ must only be a stable model of $\overline{\Pi_{\mathcal{I}}}$. That is, a stable model of the rules satisfied by $\mathcal{I}$.

The weights in a program can either be interpreted as *rewards* for satisfying rules, or as *penalties* for failure to satisfy rules (Lee, Talsania, and Y. Wang 2017b). In the reward interpretation, we can define the *unnormalized weight* $W_\Pi(\mathcal{I})$[7] of an interpretation $\mathcal{I}$ with respect to an LP$^{\text{MLN}}$ program Π as:

$$W_\Pi(\mathcal{I}) = \begin{cases} \exp\left(\sum\limits_{w:R\in\Pi_{\mathcal{I}}} w\right) & \text{if } \mathcal{I} \in SM[\Pi] \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where $\Pi_{\mathcal{I}}$ denotes the set of weighted rules $(w : R)$ such that $\mathcal{I} \models R$. The *normalized* weight $P_\Pi(\mathcal{I})$ under an interpretation is defined as:

$$P_\Pi(\mathcal{I}) = \lim_{\alpha\to\infty} \frac{W_\Pi(\mathcal{I})}{\sum_{\mathcal{J}\in SM[\Pi]} W_\Pi(\mathcal{J})}. \tag{2.5}$$

As we will see in Section 2.3.1, these normalized weights satisfy the Kolmogorov axioms of probability and define a (finite) probability structure. Intuitively, $P_\Pi(\mathcal{I})$ indicates the likelihood of $\mathcal{I}$ as a stable model; the weights can be seen as rewards for satisfying rules.

As mentioned, another interpretation of the weights, put forward in (Lee, Talsania, and Y. Wang 2017b), is to view them as *penalties* for *not* satisfying rules in Π. That is:

$$W_\Pi^{Pen}(\mathcal{I}) = \begin{cases} \exp\left(-\sum\limits_{w:R\in\Pi \text{ and } \mathcal{I}\not\models R} w\right) & \text{if } \mathcal{I} \in SM[\Pi] \\ 0 & \text{otherwise} \end{cases}.$$

[7] As discussed in Section 2.3.1, interpretations are the basic elements of S in $(S, \mathcal{A}, P_\Pi)$; as such, writing $W_\Pi(\mathcal{I})$ is a slight abuse of notation. Correct would be $W_\Pi(\{\mathcal{I}\})$ and for the normalized weights $P_\Pi(\{\mathcal{I}\})$.

The normalized weights of an interpretation $\mathcal{I}$ is given in the same way as in the reward definition:

$$P_{\Pi}^{Pen}(\mathcal{I}) = \lim_{\alpha \to \infty} \frac{W_{\Pi}^{Pen}(\mathcal{I})}{\sum_{\mathcal{J} \in SM[\Pi]} W_{\Pi}^{Pen}(\mathcal{J})}.$$

These two definitions are proportional to one another and the probabilities of the two approaches are identical:

Proposition 2. *(Lee, Talsania, and Y. Wang 2017b, pp. 19–20) For any interpretation $\mathcal{I}$:*

$$W_{\Pi}(\mathcal{I}) \propto W_{\Pi}^{Pen}(\mathcal{I}) \text{ and } P_{\Pi}^{Pen}(\mathcal{I}) = P_{\Pi}(\mathcal{I}).$$

This can be seen by rewriting $W_{\Pi}(\mathcal{I})$ as

$$\exp\left(\sum_{w:R \in \Pi} w - \sum_{w:R \in \Pi \text{ and } \mathcal{I} \not\models R} w\right) = \exp\left(\sum_{w:R \in \Pi} w\right) \cdot \exp\left(-\sum_{w:R \in \Pi \text{ and } \mathcal{I} \not\models R} w\right).$$

Here the first term of the product corresponds to the weights of all rules in Π, and the second is exactly the weight of $\mathcal{I}$ under the penalty interpretation. For $P_{\Pi}(\mathcal{I})$, the first term cancels out through the normalization factor Z, and so $P_{\Pi}(\mathcal{I}) = P_{\Pi}^{Pen}(\mathcal{I})$. As discussed in Section 2.3.3, the penalty approach has the benefit of easier translation into ASP with weak constraints.

P_{Π} can also be defined over propositions, by summing over interpretations that satisfy that proposition:

$$P_{\Pi}(A) = \sum_{\mathcal{I}:\mathcal{I} \models A} P_{\Pi}(\mathcal{I}).$$

As in Section 2.2.2, if we assign the same weights to all rules in an LP$^{\text{MLN}}$-program, the highest probability is obtained by any interpretation that satisfies the most number of rules. This corresponds then to $MAXSAT$, or maximum satisfiability problem: the problem of satisfying the highest number of clauses (or determining the maximum number that can be satisfied).

2.3.1 Weights as Probabilities

Given a signature σ, the set of all interpretations S over σ together with the normalized weight-measure P_{Π} and an algebra[8] $\mathcal{A}$ over S defines a probability space $(S, \mathcal{A}, P_{\Pi})$. For simplicity we will here take $\mathcal{A}$ to be the powerset of the interpretations S. Note that in this section we divert from the rest of this work in notation, according to usage in the literature.

[8] Since we restrict ourselves here to a signature σ with no function symbols of positive arity, there are only finitely many Herbrand interpretations. We therefore need not consider countable unions over interpretations, and do not need a Sigma-algebra.

The normalized weights P_Π are thus measures over sets of interpretations, and satisfy the Kolmogorov axioms of probability (over finite $\mathcal{A}$):

Proposition 3.

- *For any set of interpretations* $A \in \mathcal{A}$, $P_\Pi(A) \geq 0$
- $P_\Pi(S) = 1$
- *For two* $A, B \in \mathcal{A}$ *such that* $A \cap B = \emptyset$, $P_\Pi(A \cup B) = P_\Pi(A) + P_\Pi(B)$

Proof.

1. Immediate from the definition of $W_\Pi(A) = \exp\left(\sum_{w:R\in\Pi_\mathcal{I}} w\right)$ and the fact that the exponential function is non-negative.
2. Immediate: $P_\Pi(S) = P_\Pi(\sum_{\mathcal{J}\in SM[\Pi]} W_\Pi(\mathcal{J})) = \frac{\sum_{\mathcal{J}\in SM[\Pi]} W_\Pi(\mathcal{J})}{\sum_{\mathcal{J}\in SM[\Pi]} W_\Pi(\mathcal{J})} = 1$
3. Suppose A, B are of the form $\{\mathcal{I}\}, \{\mathcal{J}\}$ for some $\mathcal{I}, \mathcal{J}$. Then $W_\Pi(\{\mathcal{I}\} \cup \{\mathcal{J}\}) = W_\Pi(\{\mathcal{I}\}) + W_\Pi(\{\mathcal{J}\})$, so that $P_\Pi(\{\mathcal{I}\} \cup \{\mathcal{J}\}) = P_\Pi(\{\mathcal{I}\}) + P_\Pi(\{\mathcal{J}\})$. The case where A, B are general sets follows by simple induction.

□

The probability distribution here is over stable models of maximal subsets of Π, and $P_\Pi(\{\mathcal{I}\})$ indicates how likely it is to "draw" some interpretation, given a program Π. Intuitively, the interpretations can be seen as (complete descriptions of) possible worlds, and the probability distribution is over sets of these possible worlds.[9]

In the following we will again use $P_\Pi(\mathcal{I})$ instead of the more proper $P_\Pi(\{\mathcal{I}\})$ for convenience.

[9]For a simple fragment of LP$^{\text{MLN}}$, where the uncertainty is restricted to "probabilistic constants", which are random variables that describe a distribution over the value of a constant symbol, probability can be represented in a more natural way. Namely, the probability of an interpretation (that is a consistent stable model) here corresponds directly to the product of the probability of the constant declarations in the interpretation. The probability can therefore be calculated on the basis of the probabilistic constant declarations in the program alone. This is very similar to the way probability in Problog is defined; see (Lee and Y. Wang 2016a, p. 150) for details and also Section 2.4.

2.3.2 Modelling in LP$^{\text{MLN}}$

As defined above, the interpretations in $SM[\Pi]$ might still violate some rules with infinite weights in Π. To see this, consider the following example (adapted from (Lee and Y. Wang 2016a, p. 147)):

Program 2.8

$$\begin{aligned} &Bird(x) \leftarrow ResidentBird(x) && : 1.5 \\ &Bird(x) \leftarrow MigratoryBird(x) && : 2.5 \\ &\leftarrow ResidentBird(x), MigratoryBird(x) && : \alpha \\ &ResidentBird(Tweety) && : \alpha \\ &MigratoryBird(Tweety) && : \alpha \end{aligned}$$

Here the Herbrand universe is $\{Tweety\}$. There can be no interpretation that satisfies all rules of this program, because of the constraint (3rd rule) together with the 4th and 5th rule. Instead, some of the interpretations with weights are:

$$\{R(Tweety)\}e^{2\alpha} \tag{2.6}$$
$$\{R(Tweety), B(Tweety)\}e^{2\alpha+1.5} \tag{2.7}$$
$$\{M(Tweety), B(Tweety)\}e^{2\alpha+2.5} \tag{2.8}$$
$$\{R(Tweety), M(Tweety), B(Tweety)\}e^{2\alpha+2.5+1.5} \tag{2.9}$$

As we can see, the inconsistent interpretation that violates the third rule is actually the most probable interpretation. This might be considered undesirable, given that these hard rules can be seen as encoding definite knowledge. Let us denote the set of hard rules of Π as Π^{hard}, and the set of soft rules as Π^{soft}. To avoid violations of definite knowledge in LP$^{\text{MLN}}$ programs, we can define:

$$SM'[\Pi] = \left\{ \mathcal{I} \mid \mathcal{I} \text{ is a stable model of } \overline{\Pi_{\mathcal{I}}} \text{ that satisfy } \overline{\Pi^{\text{hard}}} \right\},$$

and define the weights as follows:

$$W'_{\Pi}(\mathcal{I}) = \begin{cases} \exp\left(\sum_{w:R \in (\Pi^{\text{soft}})_{\mathcal{I}}} w \right) & \text{if } \mathcal{I} \in SM'[\Pi] \\ 0 & \text{otherwise} \end{cases},$$

$$P'_{\Pi}(\mathcal{I}) = \frac{W'_{\Pi}(\mathcal{I})}{\sum_{\mathcal{J} \in SM'[\Pi]} W'_{\Pi}(\mathcal{J})}.$$

Here the weights, and therefore the probabilities, of interpretations can be calculated by looking at the weights of the soft rules only, if $SM'[\Pi]$ is not empty. The hard rules can then be interpreted as definite knowledge. The following proposition relates P'_{Π} and $P_{\Pi}(\mathcal{I})$:

Proposition 4. *(Lee and Y. Wang 2016b, pp. 12–13) If $SM'[\Pi]$ is not empty, $P'_\Pi(\mathcal{I})$ coincides with $P_\Pi(\mathcal{I})$.*

However, in the case where there is no interpretation that is a stable model of Π^{hard}, so that $SM'[\Pi]$ is empty, $P'_\Pi(\mathcal{I})$ is undefined. This is then a choice of how to model a situation; whether the hard rules can be violated or not.

LP$^{\text{MLN}}$ and MLN

The flexible way LP$^{\text{MLN}}$ can deal with inconsistent knowledge bases makes it easy to combine different knowledge bases: even if the resulting knowledge base is inconsistent, the LP$^{\text{MLN}}$ formalism and corresponding computations will still yield interpretations with positive probabilities. So, even if the hard rules taken together are inconsistent—as in the example above—under the initial unnormalized weight function W_Π (and corresponding probability function) inconsistencies can be easily handled. Under the second proposed semantics using the W'_Π function, this is only the case if Π^{hard} is consistent.

LP$^{\text{MLN}}$ extends ASP in a probabilistic way, as MLN extends first-order logic in a probabilistic way. To see how LP$^{\text{MLN}}$ provides an easier and more compact representation than MLN, let us reconsider our previous toy-example of how smoking influences other people:

$$
\begin{aligned}
&Smokes(y) \leftarrow Smokes(x) \wedge Influences(x, y) && : w\\
&Smokes(Alice) && : \alpha\\
&Influences(Alice, Bob) && : \alpha\\
&Influences(Bob, Carol) && : \alpha
\end{aligned}
$$

for any positive number w.

Restricting ourselves to interpretations that satisfy all hard rules and without considering the interpretation of the $Influences$ relation (which is satisfied by all interpretations), the following are four interpretations of interest, along with their weights:

$$
\begin{aligned}
&\{Smokes(Alice), \neg Smokes(Bob), \neg Smokes(Carol)\} && e^{3\alpha} \cdot e^{8w}\\
&\{Smokes(Alice), Smokes(Bob), \neg Smokes(Carol)\} && e^{3\alpha} \cdot e^{8w}\\
&\{Smokes(Alice), Smokes(Bob), Smokes(Carol)\} && e^{3\alpha} \cdot e^{9w}\\
&\{\neg Smokes(Alice), Smokes(Bob), Smokes(Carol)\} && 0
\end{aligned}
$$

The 8 and 9 are obtained by going through all groundings of the soft rule. The fourth interpretation is not a stable model of $\Pi_\mathcal{I}$: the stable model of $\Pi_\mathcal{I}$ corresponds to letting none of Alice, Bob and Carol smoke in the interpretation. As such, it has weight 0. The third interpretation obtains a higher weight by the recursion: after the rule is satisfied

with Bob substituted for y and Alice for x, it is then also applied to Carol for y and Bob for x. As a result we obtain the following probabilities:

$$P_{\Pi}(Smokes(Bob)) = \frac{e^{3\alpha+8w} + e^{3\alpha+9w}}{e^{3\alpha+9w} + e^{2*(3\alpha+8w)}}$$

$$P_{\Pi}(Smokes(Carol)) = \frac{e^{3\alpha+9w}}{e^{3\alpha+9w} + e^{2*(3\alpha+8w)}}$$

In the Markov Logic semantics the situation is different. Here the recursion cannot easily be taken into account because of the semantics derived from first-order logic. As a result, interpretations that satisfy $Smokes(Alice)$ do not also necessarily satisfy $Smokes(Bob)$. We obtain the following situation, according to Eq. (2.3):

$$P_{\mathbb{L}}(Smokes(Bob)) = \frac{e^{8w} + e^{9w}}{3e^{8w} + e^{9w}} = P_{\mathbb{L}}(Smokes(Carol)).$$

Because of this, $Smokes(Carol)$ has two interpretations of which one does not satisfy $Smokes(Bob)$, and so has the same unnormalized weight and probability as $Smokes(Bob)$. MLN therefore fails to capture the meaning of the first formula properly. This meaning and the correct probability can be captured as we shall see later, but at the expense of possibly having to introduce exponentially many formulas.

However, the result does hold in a straightforward manner in the opposite direction. Let $w : F$ be any weighted formula in a MLN program $\mathbb{L}$. If the signature σ is finite, we can eliminate any existential quantification in F by replacing it with a disjunction of all of its groundings. Alternatively, existentially quantified predicates can be replaced by fresh auxiliary predicates (Cabalar 2009). It can then be translated to an equivalent clausal form; without loss of generality, assume F therefore to be already in clausal form. Then we can turn $w : F$ in $\mathbb{L}$ into the weighted rule:

Program 2.9

$$\bot \leftarrow \neg F \qquad : w$$
$$\{A\}^{ch} \qquad : w'$$

for every ground atom A of the signature σ. Here the weighted choice rules all have the same arbitrary weight w'; this choice rule (see. Program 2.7) has the effect that any ground atom A cannot be minimized under the stable model semantics described above (Lee and Y. Wang 2016a, p. 148). We then have the following result.

Proposition 5. *(Lee and Y. Wang 2016b, p. 15)*

Any Markov Logic Network $\mathbb{L}$ and its LP^{MLN} representation $\Pi_{\mathbb{L}}$, as described in Program 2.9, have the same probability distribution over all interpretations $\mathcal{I}$.

Proof. For the proof we will assume F to be a ground formula. For convenience of representation we will consider any arbitrary (finite) signature σ as given and simply write $P_{\mathbb{L}}$ as shorthand for $P_{M_{\mathbb{L},C}}$. Similar to $\Pi_{\mathcal{I}}$, we use $\mathbb{L}_{\mathcal{I}}$ to denote the set of formulas in $\mathbb{L}$ that are satisfied by $\mathcal{I}$. We write $w : F \in \mathbb{L}_{\mathcal{I}}$ for the weighted formulas $w : F$ such that $F \in \mathbb{L}_{\mathcal{I}}$. Assume, as in Proposition 1, that some formulas can be given weights that go to infinity; denote the infinite weight with α. Denote with $At(\sigma)$ the set of all ground atoms that can be constructed using symbols from σ. We have:

$$P_{\mathbb{L}}(\mathcal{I}) = \lim_{\alpha \to \inf} \frac{\exp\left[\sum_{w:F \in \mathbb{L}_{\mathcal{I}}} w\right]}{\sum_{\mathcal{J} \in \mathfrak{I}} \exp\left[\sum_{w:F \in \mathbb{L}_{\mathcal{J}}} w\right]} = \lim_{\alpha \to \inf} \frac{\exp\left[|At(\sigma)| \cdot w'\right] \exp\left[\sum_{w:F \in \mathbb{L}_{\mathcal{I}}} w\right]}{\exp\left[|At(\sigma)| \cdot w'\right] \sum_{\mathcal{J} \in \mathfrak{I}} \exp\left[\sum_{w:F \in \mathbb{L}_{\mathcal{J}}} w\right]}$$

This corresponds to the set of atoms of the choice formulas and the translation formulas in LP$^{\mathrm{MLN}}$, meaning that the nominator can be written as

$$\exp\left[\sum_{w:F \in \mathbb{L}_{\mathcal{I}} \cup Choice(At(\sigma))} w\right]$$

and similar for the denominator, where $Choice(At(\sigma))$ stands for the set of weighted choice rules over $At(\sigma)$, which is a set of tautologies. This corresponds then simply to those atoms that are true in $\mathcal{I}$. In this way, $\Pi_{\mathbb{L}}$ satisfies the same atoms according to the second rule of Program 2.9, and $\Pi_{\mathbb{L}}$ corresponds to $\mathbb{L}_{\mathcal{I}} \cup Choice(At(\sigma))$ for any $\mathcal{I}$ by definition of Program 2.9. So we have:

$$P_{\mathbb{L}}(\mathcal{I}) = \lim_{\alpha \to \inf} \frac{\exp\left[\sum_{w:F \in (\Pi_{\mathbb{L}})_{\mathcal{I}}} w\right]}{\sum_{\mathcal{J} \in \mathfrak{I}} \exp\left[\sum_{w:F \in (\Pi_{\mathbb{L}})_{\mathcal{J}}} w\right]}.$$

It then still needs to be shown is that the stable models of $\overline{(\Pi_{\mathbb{L}})_{\mathcal{I}}}$ are the models of $\overline{\mathbb{L}_{\mathcal{I}}}$. We will not prove this here, but this can be seen by the fact that any Herbrand interpretation of $\overline{\mathbb{L}_{\mathcal{I}}}$ assigns a truth value to every possible ground atom over σ, and that the choice rules in $\Pi_{\mathbb{L}}$ also have this effect. Since any $\mathcal{I}$ is itself a model of $\overline{\mathbb{L}_{\mathcal{I}}}$, and therefore also of $\overline{(\Pi_{\mathbb{L}})_{\mathcal{I}}}$, we have that $P_{\mathbb{L}}(\mathcal{I}) = P_{\Pi_{\mathbb{L}}}(\mathcal{I})$ for any $\mathcal{I}$. □

Contrary to what might appear to be the case from the previous example, given our restriction to σ, every LP$^{\mathrm{MLN}}$ program does in fact have a corresponding Markov Logic network. To show this, we need a few more notions.

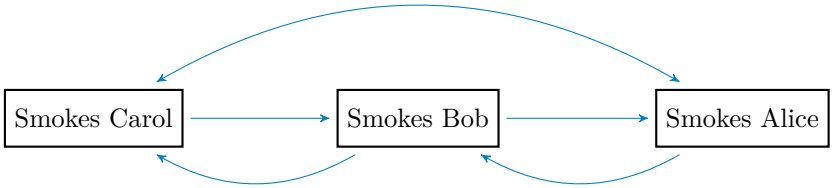
Let us denote by $posat(R)$ all the atoms A in a rule R such that at least one occurrence of A is not within the scope of default negation *not*. Then for any logic program Π, the *positive dependency graph* G_{Π} is the directed graph constructed as follows:

- For any atom A in Π there is a node v_A;
- For every rule $Head \leftarrow Body$ in Π, there is an edge e_{a_1,a_2} from v_{A_2} to v_{A_1} iff $v_{A_2} \in H$ and $v_{A_1} \in posat(Body)$. Thus, there is an edge from every atom in $Head$ to every positive atom in $Body$.

Here v_A is the vertex in the graph corresponding to atom A.

Then a logic program is called *absolutely tight* if its positive dependency graph has no cycles.

For example, the positive dependency graph of our smoking example, not showing the $Influences$ relations for easier presentation, looks like this:



and clearly contains cycles. It is therefore not absolutely tight.

Absolutely tight LP$^{\text{MLN}}$ programs have a rather straightforward translation to equivalent Markov Logic Networks (Lee and Y. Wang 2016a, p. 149). For this we define the *completion formulas* of Π, $Comp(\Pi)$.

Let Π be any logic program with rules of the form

$$Head \leftarrow B_1, ..., B_k, not\ B_{k+1}, not\ B_n$$

Then the $Comp(\Pi)$ is defined as the set of formulas:

- $Body \rightarrow Head$;
- $A \rightarrow \bigvee_{\substack{(Head \leftarrow Body) \in \Pi \\ A \in Head}} (Body \wedge \bigwedge_{A' \in Head \setminus \{A\}} \neg A')$, for every atom A.

An important result for classical logic programs is that for any absolutely tight logic program Π, a set of atoms $\{A_1, ..., A_n\}$ satisfies Π if and only if $\{A_1, ..., A_n\}$ satisfies the completion of Π, $Comp(\Pi)$ (Lee and Lifschitz 2003). This result extends to LP$^{\text{MLN}}$ programs:

Proposition 6. *(Lee and Y. Wang 2016a, p. 149) For any tight absolutely tight LP^{MLN} program Π, if $SM[\Pi]$ is not empty, Π under the LP^{MLN} semantics and $Comp(\Pi)$ under the MLN semantics have the same probability distribution over all interpretations.*

This notion of program completion can be generalized by introducing *loop formulas*. Define a *loop* in a program Π, denoted L, as a non-empty set of atoms such that for any pair of atoms $A_1, A_2 \in L$, there exists a path of non-zero length from v_{A_1} to v_{A_2} in the positive dependency graph of Π using only edges connecting elements of L.

By this definition, our previous example contains four loops:

$$L_1 = \{Smokes(Carol), Smokes(Bob)\},$$
$$L_2 = \{Smokes(Bob), Smokes(Alice)\},$$
$$L_3 = \{Smokes(Carol), Smokes(Alice)\},$$
$$L_4 = \{Smokes(Carol), Smokes(Bob), Smokes(Alice)\}.$$

Then for any loop L, denote by $R(L)$ the set of formulas:

$$B \wedge \bigwedge_{p \in H \setminus L} \neg p$$

for all rules in Π such that $Head \cap L \neq \emptyset$ and $posat(Body) \cap L = \emptyset$. Then denote by $CLF(L)$ the *Conjunctive loop formula* of L:

$$CLF(L) = \bigwedge L \rightarrow \bigvee R(L).$$

Denote the set of all CLF formulas of a program Π with $CLF(\Pi)$. The important result from Lee and Lifschitz is then:

Proposition 7. *(Lee and Lifschitz 2003, p. 457) For any Logic Program* Π *of the form Eq.* (2.1) *and any set* X *of atoms, the following are equivalent:*

- X *is an answer set of* Π*,*
- X *is a model of* $Comp(\Pi) \cup CLF(\Pi)$*.*

For our previous example, adding the loop formula

$$(Smokes(Alice) \wedge Smokes(Carol)) \rightarrow (Influences(Alice, Carol) \vee Influences(Carol, Alice) \vee Smokes(Bob))$$

(the other loop formulas are trivially satisfied due to the hard facts given), we obtain an equivalent Markov Logic Network. These results are important for the topic of weight learning, as will be seen in the next chapter.

The benefit of LP$^{\mathrm{MLN}}$ over MLN is not just the easier representation of these recursive programs, but also the number of formulas needed for the representation and the computation. The problem with the loop formulas translation is that a program can have exponentially many loops and therefore exponentially many loop formulas might be needed (Lin and Zhao 2004; Lee and Lifschitz 2003).

Essential for this translation is the requirement that we have no function symbols of positive arity. As already remarked, ASP is more expressive than first-order logic; the construction of loop formulas is not restricted to LP$^{\mathrm{MLN}}$ but applies to ASP in general, and not every rule of a logic program can be rewritten to an equivalent (under standard first-order semantics) first-order formula (Lee and Meng 2011). In the case that the signature is restricted as in this thesis, the result does hold.

2.3.3 LP$^{\text{MLN}}$ and ASP

Many solvers for ASP also support weak constraints; if there was a translation of LP$^{\text{MLN}}$ programs to ASP programs with weak constraints, inference in LP$^{\text{MLN}}$ could be done by a translation to the input language of these solvers. It turns out that rather easy and efficient translations exist. Translations to and from ASP with weak constraints can be done for both the reward and penalty interpretations of LP$^{\text{MLN}}$, as described above in Section 2.3. We will consider both ways in turn, after considering first the translation from weak constraints to LP$^{\text{MLN}}$.

A translation from ASP with weak constraints to LP$^{\text{MLN}}$ is straightforward. Any hard rule or constraint is simply given the weight α. Take any constraint of the following form:

Program 2.10

$$:\sim B_1, ..., B_k, not\ B_{k+1}, ..., not\ B_n \qquad : w$$

Then this can be translated into the LP$^{\text{MLN}}$ rule as:

Program 2.11

$$\leftarrow not\ B_1; ...; not\ B_k; not\ not\ B_{k+1}, ..., not\ not\ B_n \qquad : -w$$

The idea here is that weak constraints provide penalties and are most naturally interpreted as minimization problems: the answer set with minimal overall penalties is optimal. For LP$^{\text{MLN}}$ we capture this by a negative translation, whose rules are satisfied exactly when the body (of the weak constraint) can be assumed to be *false*; in this case a negative weight is added corresponding to a negative reward, which is essentially a penalty.

Reward interpretations

For the purpose of computing LP$^{\text{MLN}}$, the translation the other way is very important: this allows for the usage of ASP solvers for computing LP$^{\text{MLN}}$ queries. This translation is more involved. For this, introduce for the i-th rule, with weight w_i and global variables x, the auxiliary atom $sat(i, w_i, x)$, whose value is $True$ if the corresponding ground atom from the LP$^{\text{MLN}}$ program is true. Then, letting again $Body$ denote the entire (negative or positive) body of an LP$^{\text{MLN}}$ rule, and $Head$ the entire head, we rewrite any LP$^{\text{MLN}}$ rule of the form:

$$Head(x) \leftarrow Body(x) \qquad : w_i$$

whose index is i with occuring global variables x, into the following set of ASP rules:

Program 2.12

$$sat(i, w_i, x) \leftarrow Head(x)$$
$$sat(i, w_i, x) \leftarrow not\ Body(x)$$
$$Head(x) \leftarrow Body(x), not\ not\ sat(i, w_i, x)$$
$$:\sim sat(i, w_i, x). \qquad [-w_i'@l]$$

Where $w_i' = 1$ and $l = 1$ if w_i is α; and $w_i' = w_i$ and $l = 0$ otherwise.

This has the effect that $sat(i, w_i, x)$ is true if either $Head$ is true, or $Body$ is not true in the interpretation (can be assumed to be false). For non-hard rules, the $sat(i, w_i, x)$ atom then receives the negative weight $-w_i'$ as a penalty at level 0, which is essentially a reward. For hard rules (with weight α), w_i' is set to -1 at level 1 (Lee, Talsania, and Y. Wang 2017b, p. 4). (Recall that higher levels are optimized first.)

The problem with the translation just described is that it does not necessarily give an acceptable ASP program for an ASP solver. First, $Head(x)$ might involve disjunction, which might not be allowed in the body of rules, depending on the ASP solver. In addition, $not\ Body(x)$ will have to be rewritten to be allowed in the input language. More importantly, the first and second rule of the translation can be unsafe according to Definition 1, so that grounding the problem might not be possible in ASP solvers (Lee, Talsania, and Y. Wang 2017b, p. 5).

Penalty translation

It turns out that a less problematic translation is possible using the (equivalent) penalty interpretation of weights as given in Section 2.3. For any rule R:

$$Head(x) \leftarrow Body(x) \qquad : w_i$$

we can translate R according to the penalty semantics into the following ASP rules with weak constraints:

Program 2.13

$$unsat(i, w_i, x) \leftarrow Body(x), not\ Head(x)$$
$$Head(x) \leftarrow Body(x), not\ unsat(i, w_i, x)$$
$$:\sim unsat(i, w_i, x). \qquad [w_i'@l]$$

where w_i' is as before in the reward interpretation.

In the case $Head(x)$ is a disjunction $A_1; ..., A_m$, $not\ Head(x)$ stands for $not\ A_1, ..., not\ A_m$. It is clear that all rules of this translation are safe. Furthermore, the resulting set of rules belongs to the input language of standard ASP solvers.

In this translation, *unsat* is true when the i-th rule is not satisfied. If $unsat(i, w_i, x)$ is true, w_i is imposed as the penalty on the stable model. If the i-th rule is satisfied, $unsat(i, w_i, x)$ is false.

We refer to this translation as lpmln2asp^{pnt}. We then have the following result:

Proposition 8. *(Lee, Talsania, and Y. Wang 2017b, p. 6)*
For any LP^{MLN} program Π*, there is a 1-1 correspondence* Φ *between* $SM[\Pi]$ *and the set of stable models corresponding to lpmln2asp^{pnt}*(Π)*. Furthermore,* Φ *is a 1-1 correspondence between the most probable stable models of* Π *and the optimal stable models of lpmln2asp^{pnt}*(Π)*.*

This provides a way to compute LP$^{\text{MLN}}$ using ASP solvers that allow weak constraints in the input, while also ensuring that the translated program is part of the input syntax of ASP solvers (such as Clingo).

2.4 Related Formalisms

There are multiple formalisms in the ASP paradigm that are related to LP$^{\text{MLN}}$ and also allow to express uncertainty in some way. These formalisms are of interest by themselves; however, because of space constraints, we will here mostly restrict ourselves to a formalism that is related to weight learning in LP$^{\text{MLN}}$, the topic of this thesis: ProbLog. Other important related formalisms that should be mentioned here are P-Log (Baral, Gelfond, and Rushton 2009), and the already discussed ASP with weak constraints.

As we have seen in Section 2.3.3, any program of ASP with weak constraints can be turned into an equivalent LP$^{\text{MLN}}$ program, and any LP$^{\text{MLN}}$ program can be encoded in ASP with weak constraints. Furthermore, from Proposition 8 and as is discussed in detail in (Lee and Yang 2017), there is a 1-1 correpondence between the most probable stable models and the optimal stable models in the translated program with weak constraints. The benefit of LP$^{\text{MLN}}$ is the ease of modelling and representation, plus the intuitive probabilistic semantics.

Before looking in slightly more detail into the relationship of LP$^{\text{MLN}}$ with ProbLog, we will now give a broad overview of P-Log.

2.4.1 P-Log

P-Log also provides a probabilistic semantics for ASP. It does this by adding probabilistic syntactic constructs to a P-Log program definition, where the probabilies can loosely be understood as a measure on an agent's degree of belief on an atom (Baral, Gelfond, and Rushton 2009). A P-Log program can be divided clearly into a *logical* and a *probabilistic* part: the logical part represents knowledge and consists of standard ASP rules along with declarations of random attributes, while the probabilistic part contains "pr-atoms", or *causal probability statements* (Balai and Gelfond 2016, p. 4), which determine the

probability of possible worlds (corresponding to answer sets in the above sections) (Baral, Gelfond, and Rushton 2009, pp. 3–4).

Describing the syntax and semantics of P-Log in detail is beyond the scope of this thesis; of importance here is a broad understanding of the probabilistic semantics. A P-Log program consists of six parts[10]: (i) a sorted signature, (ii) a declaration, (iii) a regular part, (iv) a set of random selection rules, (v) a probabilistic information part, and (vi) a set of observations and actions (Baral, Gelfond, and Rushton 2009, p. 6). The uncertainty in P-Log lies in the set of random selection rules (iv) and the probabilistic atoms (v). A random selection rule is of the form:

$$[r]\ random(f(\bar{t}) : \{x : p(x)\}) \leftarrow Body.$$

Here $\bar{t}$ is a vector of terms. This intuitively says that if $Body$ holds, then the value of $f(\bar{t})$ is chosen at random from $\{x : p(x)\} \cap range(f)$ by experiment r, unless $f(\bar{t})$ is specified already by some action (in (vi)). A probabilistic atom is of the form:

$$pr_r(f(\bar{t}) = a \mid_c Body) = p.$$

Here r is the name of some random selection rule, $p \in [0, 1]$, and $Body$ is a set of literals of (i), the signature. This says that if the value of $f(\bar{t})$ is fixed by experiment r and $Body$ holds, then the probability that r causes $f(\bar{t}) = a$ is p.

In a P-Log program, due to the random selection rules and probabilistic atoms, given two conditions called the *unique random selection rule* and *unique probability assignment*, possible worlds can be assigned measures which are probabilities (Lee and Yang 2017, p. 1174; Baral, Gelfond, and Rushton 2009, pp. 11, 15–16).

As shown in (Lee and Yang 2017), P-Log can be translated to LP$^{\text{MLN}}$ and in that way (through our connection with ASP with weak constraints) inference can be done using ASP solvers. Conversely, as shown in (Balai and Gelfond 2016), LP$^{\text{MLN}}$ can be encoded in P-Log in linear time (in the size of the LP$^{\text{MLN}}$ program). The difference between the two formalisms lies in the highly structured semantics and syntax of P-Log. In P-Log, probabilities are assigned to atoms only, which is intuitive from a specific modeling perspective but more complex from the general situation where rules are given a ranking based on importance, or weights, as opposed to specifying the probabilities of atoms. The weights in LP$^{\text{MLN}}$ are more complex to assign because of the looser structure, but the language is less involved and provides an easier probabilistic expansion upon ASP. From a weight learning perspective, it is also simpler to analyze. Because of the higher conceptual complexity of the relation between LP$^{\text{MLN}}$ and P-Log, P-Log will not be further discussed in this thesis.

[10] Because we do not discuss the syntax in detail, we gloss over the explanation of some terms in this section to keep the explanation short. See (Balai and Gelfond 2016; Lee and Yang 2017; Baral, Gelfond, and Rushton 2009) for a complete account.

2.4.2 ProbLog

A ProbLog program[11] $\langle PF, \Pi \rangle$ consists of a set of ground probabilistic facts PF, and a set of rules and non-probabilistic facts Π (a normal logic program, as introduced in Section 2.1[12]). As the non-probabilistic parts have already been explained, we will only focus on the probabilistic facts. A probabilistic fact is a ground fact f annotated with a probability p, written $p :: f$. An atom that corresponds to a grounding of some probabilistic fact is called a *probabilistic atom*.[13] Here the set of probabilistic atoms must be disjoint from the set of derived atoms (i.e., atoms in the head of some rule of Π).

ProbLog provides a probabilistic semantics for ASP using Sato's distribution semantics (Sato 1995). We will restrict ourselves to a finite Herbrand base. Each ground probabilistic fact in ProbLog allows an *atomic choice*, such that we can either choose to include f with probability p, or to not include it with probability $1 - p$. A *total choice* consists of an atomic choice of all probabilistic atoms (probabilistic ground facts). Then the probability of a total choice is straightforwardly defined as the product of all the atomic choices, which follows because all the atomic choices correspond to independent events.

The general semantics of ProbLog differs from that of LP^{MLN} and ASP, and is based on the *well-founded semantics* (Van Gelder, Ross, and Schlipf 1991), which will not be discussed here. Roughly, a well-founded model is defined as the fixed-point of an operator that (iteratively) only includes well-founded literals. In the case of a well-founded *total* model of a logic program it holds that this model corresponds to the unique stable model of the program (Van Gelder, Ross, and Schlipf 1991, p. 633). However, there can be unique stable models without there being a total well-founded model of some progam.

A Herbrand interpretation $\mathcal{I}$ is said to be a *model* for a ProbLog program if there exists a total choice C, such that for the well-founded model WFM of $C \cup \Pi$, denoted $WFM(C \cup \Pi)$, it it is the case the $WFM(C \cup \Pi) = \mathcal{I}$, where Π denotes the (non-probabilistic) normal logic program of the ProbLog program. It is important here that the ProbLog semantics is only defined for programs that are *sound*: programs for which each possible total choice leads to a unique well-founded Herbrand interpretation that satisfies all ground rules of the ProbLog program (is *total*)(Fierens et al. 2015, p. 364). The probability of a Herbrand interpretation is then said to be the probability of its total choice if it is a model of the ProbLog program, otherwise it has probability 0.

This highlights the important difference with LP^{MLN}: whereas ProbLog is only well defined if every total choice leads to a unique well-founded model, LP^{MLN} can handle multiple stable models. This makes LP^{MLN} into a proper generalization of ProbLog (Lee and Y. Wang 2016a, p. 149).

[11] In this section we follow the explanation of (Fierens et al. 2015).

[12] ProbLog does not allow disjunction in the head, nor double default negation in the body.

[13] ProbLog also allows probabilities assigned to head of rules, as in $0.9 :: alarm \leftarrow burglary$. These are however equivalent via a suitable translation to only allowing probabilities assigned to probabilistic facts, and for brevity we will ignore this extra construct. See (*ProbLog 2.1 documentation* 2014) for details.

The translation from ProbLog to LP$^{\text{MLN}}$ is straightforward (Lee and Y. Wang 2016a, p. 149). Namely, given any ProbLog program $\langle PF, \Pi \rangle$, construct for every probabilistic fact $p :: A$ in PF, the LP$^{\text{MLN}}$ rule:

Program 2.14

$$r = \begin{cases} \ln(p) : A \text{ and } \ln(1-p) :\leftarrow A & \text{if } 0 < p < 1 \\ \alpha : A & \text{if } p = 1 \\ \alpha :\leftarrow A & \text{if } p = 0 \end{cases}$$

In addition, every rule from Π can be straightforwardly given the weight α in the LP$^{\text{MLN}}$ program.

In addition, if all soft rules in an LP$^{\text{MLN}}$ program are of the form $w : A$, where A is a first-order atom which does not occur in the head of any rules with weight α, and if for any assignment of values to soft rules there is exactly 1 probabilistic stable model that satisfies this truth assignment, there is a corresponding ProbLog program. Call such an LP$^{\text{MLN}}$ program *1-coherent*. Then, the probabilistic fact corresponding to any such a soft rule corresponds to:

$$p :: A, \text{ where } p = \frac{\exp(w)}{1 + \exp(w)}.$$

This allows to perform weight learning on some restricted LP$^{\text{MLN}}$ programs in ProbLog, by learning the probabilities in ProbLog, and translating the learned probabilities back using the inverse procedure. Further details on the relation and translation can be found in (Lee and Y. Wang 2018).

To sum up, we have the following results:

Proposition 9. *(Lee and Y. Wang 2018, p. 6)*

- *Any well-defined ProbLog program ProbLog and the corresponding LP^{MLN} program $\Pi_{ProbLog}$ as described in Program 2.14 have the same probability distribution over all interpretations.*

- *For any 1-coherent LP^{MLN} program Π and any interpretation $\mathcal{I}$, we have*

  $\boldsymbol{w} = argmax_{\boldsymbol{w}} P_{\Pi}(\mathcal{I})$
  if and only if
  $\boldsymbol{pr} = argmax_{\boldsymbol{pr}} P_{ProbLog}(\mathcal{I})$.

2.5 Applications

As LP^{MLN} can be seen as both an extension of MLN and of traditional ASP, it can be used for many purposes. It lends itself well for statistical relational learning (SRL)[14], which it has in common with MLN, in addition to being able to model and solve complex search and optimization problems, a strong application of standard ASP. Decision problems are also a good application of LP^{MLN} because of the possibility of representing degrees of belief of an agent (Baral, Gelfond, and Rushton 2009), as well as knowledge representation (Brewka, Eiter, and Truszczyński 2011).

One particular subject to which the formalism has been successfully applied is *collective classification* (Sen et al. 2010b). Rather than try to classify objects or entities based only on so-called *local* features, collective classification makes the predicted value of the i-th instance dependent on the predicted value of related instances. By focusing only on local features of the instance itself, not in its relation to other objects, traditional classification fails to take relations among objects in the data into account. Modeling data as a collective classification problem can improve the quality of a model where this structure is important for representing the data. A few examples of this are text tagging and chunking, where knowing the value of nearby words is very useful for classification of a word, or object labeling in images. Since LP^{MLN} has been applied by Eiter and T. Kaminski for the latter (Eiter and T. Kaminski 2016), this is the application for which we will test weight learning in Chapter 4. This particular application is also well-suited for comparing different weight learning algorithms, as the prediction accuracy is a good measure of the quality of a learned model in this setting. Modelling the object-labelling problem using LP^{MLN} allows to easily take spatial relations among objects into account, which we will do in our example encoding.

[14] See (Getoor and Taskar 2007) for an extensive introduction.

CHAPTER 3

Weight Learning in LP$^{\text{MLN}}$

As a probabilistic reasoning formalism, it is vital that the weights of formulas in an LP$^{\text{MLN}}$ program accurately capture real probabilities of a dataset, so that the normalized weights of interpretations ("possible worlds") correspond to as good a representation of the actual data distribution as possible. Unless the weights are given by expert knowledge, the quality of the program as a model depends directly on the method used for learning the weights. Given a large enough and representative dataset (whose instances are independent and identically distributed), a model of the distribution can be learned; the goal of learning is to choose that learned model which best captures the distribution of our data.[1]

Given a "structure" of an LP$^{\text{MLN}}$ program — a set of rules — the problem of obtaining the most accurate settings of the weights in an LP$^{\text{MLN}}$ program can be described in terms of parameterized LP$^{\text{MLN}}$ programs: programs Π where the weights of soft rules are replaced with distinct parameters $\boldsymbol{w}$; we denote parameterized models by $P_{\Pi_{\boldsymbol{w}}}$, or by $P_{\boldsymbol{w}}$ if Π is clear from context. What is the most accurate or best model is of course dependent on the choice of a performance metric: what "best" means is not immediately clear and depends on the purpose of our learning task and on the specific problem specification. This will be important with respect to the different applications of LP$^{\text{MLN}}$ that will be considered later on.

One popular method of evaluating the performance of a model is to consider how well it predicts the data we have. This means that we are looking for the parameter settings $\hat{\boldsymbol{w}}$, such that $P_{\hat{\boldsymbol{w}}}$ gives the highest probability of observing the data we have, over all settings of $\boldsymbol{w}$. This is called *Maximum Likelihood Estimation* (MLE): of all the possible worlds

[1] A separate part of learning LP$^{\text{MLN}}$ programs is learning the rules, or structure, of the LP$^{\text{MLN}}$ program or Markov Logic Network. This problem is generally an even harder problem than learning the weights, and also requires very different techniques. In this thesis we will restrict ourselves to the weight learning task and assume the rules of any LP$^{\text{MLN}}$ program to be given.

consistent with the LP$^{\text{MLN}}$ program, we assume our data is representative enough such that it captures the actual world. The parameter settings must then make the actual dataset we have the most likely of all possibilities. The *likelihood function* $L(\boldsymbol{w} \mid \mathcal{D})$ we seek to maximize is given by:

$$L(\boldsymbol{w} \mid \mathcal{D}) = \prod_{j}^{m} P_{\boldsymbol{w}}(\mathcal{I}[j]|\boldsymbol{w})$$

for a dataset $\mathcal{D}$ of m instances, where $\mathcal{I}[j]$ denotes the interpretation given by the j-th data instance. Such a data instance, or database, is essentially a vector of atomic formulas $\mathcal{D}_j = (d_1, ..., d_k, ..., d_z)$ where d_k is the value of the k-th atom: 1 if $d_k \in \mathcal{D}_j$ (the j-th data instance), and 0 otherwise. Every such (complete)[2] data instance characterizes a Herbrand interpretation, denoted $\mathcal{I}[j]$. The maximum likelihood optimization problem can be phrased as choosing that weight setting $\boldsymbol{w}$ such that:

$$L(\hat{\boldsymbol{w}} \mid \mathcal{D}) = \max_{\boldsymbol{w} \in \boldsymbol{\Theta}} L(\boldsymbol{w} \mid \mathcal{D})$$

which is in the case of LP$^{\text{MLN}}$ equal to:

$$L(\hat{\boldsymbol{w}} \mid \mathcal{D}) = \max_{\boldsymbol{w} \in \boldsymbol{\Theta}} \prod_{j}^{m} \frac{1}{Z(\boldsymbol{w})} exp\left(\sum_{\boldsymbol{w}: R \in \Pi_{\mathcal{I}[j]}} \boldsymbol{w} \right). \tag{3.1}$$

As we shall see, optimizing the Maximum likelihood estimate is too expensive in general for LP$^{\text{MLN}}$ programs, and approximate optimization methods, or optimization of an approximation of the likelihood function, is usually required. We will discuss maximum likelihood based learning in Section 3.1, along with different ways to make the optimization (more) tractable, or ways to make the learning more successful. Frequently, we know in advance which predicates in a program are evidence and which are going to be queried, and this knowledge can sometimes be used to learn with higher efficiency or higher effectiveness; this is called *discriminative* learning, as opposed to *generative* learning, which we shall discuss in Section 3.1.1. Alternatively, specific knowledge about the uses of a model (e.g., classification tasks) can be used to optimize the learning with respect to a specific penalty function. We will discuss alternative objectives (or approximations of the standard objective), which frequently involve assumptions about the data or about uses of the LP$^{\text{MLN}}$ program, in Section 3.2.

Other important topics within weight learning involve general topics such as overfitting (Section 3.1.2), or specific data scenarios such as missing data and noisy data (Section 3.3). Overfitting is of course a general supervised learning topic, which we will not discuss in general here but only touch upon. In Section 3.1 we will also briefly visit some specific issues with overfitting. Lastly, in Section 4.3.2 we will consider an example dataset for learning in LP$^{\text{MLN}}$.

[2] We will consider incomplete databases later on in Section 3.3.3.

3.1 Maximum Likelihood Based learning

Let us consider the logarithm of the likelihood expression for LP^{MLN} as given by Eq. (3.1):

$$\ln P_\Pi(\mathcal{I}) = l(\boldsymbol{w} : \mathcal{D}) = \sum_i^n w_i n_i(\mathcal{I}) - \ln(Z(\boldsymbol{w})) \tag{3.2}$$

for a single data instance. Here $Z(\boldsymbol{w})$ is the partition function, which involves summing over all posssible interpretations:

$$Z(\boldsymbol{w}) = \sum_{\mathcal{J}\in\mathfrak{J}} \exp\left[\sum_{i=1}^n w_i n_i(\mathcal{J})\right].$$

A primary source of complexity is this partition function, which couples all the different interpretations satisfying a program. Since learning requires multiple inference steps, this partition function must be calculated multiple times in a learning procedure. This means all possible interpretations must be computed at every step of the learning procedure, which is an **NP**-hard problem. Furthermore computing the number of true groundings $n_i(\mathcal{I})$ of a rule R_i is already #**P**-complete in the length of the rule (Richardson and Domingos 2006, p. 118). We therefore have multiple sources of complexity which have to be dealt with to make learning a feasible procedure.

As we have seen in Section 2.3.2, there is a one-to-one correspondence between LP^{MLN} and Markov Logic Networks. Although the translation from LP^{MLN} to MLN might introduce exponentially many new formulas (due to the loop formulas) and is therefore frequently unfeasible, the translation from Markov Logic to LP^{MLN} consists of one rule per formula, plus a choice rule for every atom; this translation can clearly be done in polynomial time or in logspace. It follows that weight learning in LP^{MLN} is at least as hard as weight learning in Markov Logic. In MLN exactly the same complexity as described in the previous paragraph holds; however, for Markov Logic Networks (and standard Markov Networks) weight learning has been quite extensively studied, so that efficient algorithms or efficient approximations exist. Furthermore, as we have seen in Section 2.3, the equation for P_Π for any LP^{MLN} program Π is similar to Eq. (2.3). Considering weight learning in Markov Logic is therefore useful for understanding the problem for LP^{MLN}.

Let us focus more closely on the likelihood function. Because we are interested in finding the parameter settings such that the likelihood function is maximal, and since the natural logarithm is a strictly increasing function, we can optimize the *log-likelihood* instead: its maximal will be the same as the maximal of the likelihood function. Since the probability function for Markov Logic (and LP^{MLN}) is a log-linear function, optimizing the log-likelihood is much easier. The log-likelihood has the form:

$$l(\boldsymbol{w} : \mathcal{D}) = \sum_j^m \sum_i^n (w_i n_i(\mathcal{I}[j])) - m \cdot \ln(Z(\boldsymbol{w})) \tag{3.3}$$

for m data instances and n rules or formulas. Dividing both sides by the size of our dataset, m, gives the expression:

$$\frac{1}{m}l(\boldsymbol{w}:\mathcal{D}) = \sum_i^n w_i \frac{\sum_j^m (n_i(\mathcal{I}[j]))}{m} - \ln(Z(\boldsymbol{w})) = \sum_i^n w_i \boldsymbol{E}_{\mathcal{D}}[n_i] - \ln(Z(\boldsymbol{w})), \tag{3.4}$$

where $\boldsymbol{E}_{\mathcal{D}}[n_i] = \frac{1}{m}\sum_{j=1}^m [n_i(\mathcal{I}[j])]$ is the empirical expectation of n_i: the average number of times the i-th formula is true in the dataset. When trying to maximize this expression, we are trying to increase the difference between the log-measure of the data and the log-measure of all instances. This is a contrastive objective, where the second term (the partition function) is the more complex to evaluate, since it requires summing over all possible values in $\mathfrak{I}$.

An important property of the partition function $Z(\boldsymbol{w})$ is that it is a convex function of the parameters (Koller and Friedman 2009, pp. 947–48). As such, the log-likelihood function is concave (involving the negative of $Z\boldsymbol{w}$) and has no local optima. However, because of a possible redundant parameterization in the Markov Network—such that there are multiple parameterizations of a Network that lead to the same distribution[3]—there might be multiple solutions for the maximum (log-)likelihood estimation problem. It follows that there is a unique global optimum, with (many) different solutions.

Because of the convexity, we can obtain the maximum as the zeros of the derivative w.r.t. $\boldsymbol{w}$ of the log-likelihood:

$$\frac{\partial}{\partial w_i}\frac{1}{m}l(\boldsymbol{w}:\mathcal{D}) = \boldsymbol{E}_{\mathcal{D}}[n_i] - \boldsymbol{E}_{\boldsymbol{w}}[n_i] \tag{3.5}$$

This minimum, and with that the maximum likelihood estimate, is therefore obtained by minimizing the difference between the empirical expectation ($\boldsymbol{E}_{\mathcal{D}}[n_i]$) and the expected sufficient statistics of the likelihood function:

$$\boldsymbol{E}_{\boldsymbol{w}}[n_i] = \frac{1}{Z(\boldsymbol{w})}\exp\left[\sum_{i'} w_i n_i(\mathcal{I}[i'])\right] \cdot n_i(\mathcal{I}[i']) \tag{3.6}$$

which is the expected value of n_i relative to $P_{\boldsymbol{w}}$. Here the outer summation is over all possible interpretations i'. Thus, we are searching for the parameter setting such that the expected value of n_i is equal to the empirical expectation in the data.

Unfortunately, there is no analytical solution for this maximum likelihood estimate. Resorting to iterative methods (e.g., gradient ascent) is a standard method to perform the optimization in this case. For a single dataset, this means that at every iteration a step is taken in the direction of the gradient with respect to the i-th weight (of the i-th soft rule):

$$\frac{\partial \ln P_{\Pi_{\mathcal{I}}}}{\partial w_i} = -n_i(\mathcal{I}) + \sum_{\mathcal{J}\in SM[\Pi]} P_{\Pi_{\boldsymbol{w}}}(\mathcal{J}) \cdot n_i(\mathcal{J}). \tag{3.7}$$

[3] In general, for Markov Networks, or Markov random fields, there can be infinitely many parameterizations that give describe the same distribution (Koller and Friedman 2009, pp. 128–133).

The problem is that at every iteration t, $P_{\Pi^t}(\mathcal{I})$ (the probability distribution at the t-th learning iteration) would have to be calculated, which means performing inference at every step. To perform inference, the partition function has to be calculated, meaning computing all possible interpretations or stable models, which can be very costly even for very small Networks. In addition, Markov Logic Networks and LP^{MLN} introduce the extra complexity of counting the number of true groundings of a formula in an interpretation, as mentioned above.

Given that the problem of the inference steps has been tackled, or if the complexity of inference on a specific Network or program is manageable, standard gradient ascent might also be too slow to converge because of the dependency on the parameter settings. In this case a popular algorithm is the *L-BFGS* algorithm, which performs gradient ascent by using line search instead of computing the Hessian[4](Koller and Friedman 2009, p. 950).

Despite the complexity of counting the number of true groundings, which might become an issue in larger datasets, this is normally acceptable to count exactly and will not be discussed further here. A possibility would be to use sampling methods to approximate this term; in all our experiments, we count the exact number.

It is the second source of complexity of inference, computing the partition function $Z(\boldsymbol{w})$ at every iteration, which can be seen as the most problematic. One option which might make learning more feasible is to learn the model in situations where we can partition the predicates into query and evidence predicates: conditional or discriminative learning. We will consider this before moving on to methods which use approximative methods.

3.1.1 Conditional Likelihood Maximization

As already mentioned, we frequently know *a priori* which predicates will be observed and which predicates will be queried. In the case of such a particular inference task, it can be more efficient to focus on the conditional distribution. It *can*, because whether this is indeed the case depends on the particular learning task and LP^{MLN} program we want to learn the weights of. Usually, discriminative methods perform better on larger datasets, where there are only few query atoms.

In the case of discriminative learning, we partition the ground atoms into two sets $\boldsymbol{A}, \boldsymbol{B}$, a set of evidence atoms $\boldsymbol{A}$ and a set of query atoms $\boldsymbol{B}$ (Singla and Domingos 2005, p. 3). This also partitions rules according to rules involving query predicates $\boldsymbol{q}$ (at least one grounding of the rule contains a query atom), and other rules which are considered evidence (whose groundings do not contain a single query atom). Evidence predicate symbols $\boldsymbol{p}$ are predicates of which no grounding contains a query atom. We then also partition the interpretation function $\mathcal{I}$ into a part that specifies the truth value of the evidence atoms $\mathcal{I}_{\boldsymbol{A}}$ and a part that specifies the query atoms $\mathcal{I}_{\boldsymbol{B}}$. Restricting again to a

[4]The square matrix of second-order partial derivatives, which gives information of the optima of a function.

single dataset $m = 1$, the conditional likelihood $L_{\mathcal{I}_{\boldsymbol{B}}|\mathcal{I}_{\boldsymbol{A}}}$ objective corresponds to:

$$L_{\mathcal{I}_{\boldsymbol{B}}|\mathcal{I}_{\boldsymbol{A}}}(\boldsymbol{w}|\mathcal{D}) = P(\mathcal{I}_{\boldsymbol{B}} \mid \mathcal{I}_{\boldsymbol{A}} : \boldsymbol{w}) = \frac{1}{Z(\mathcal{I}_{\boldsymbol{A}} \mid \boldsymbol{w})} \exp\left[\sum_{i \in F_{\mathcal{I}_{\boldsymbol{B}}}} w_i n_i(\mathcal{I}_{\boldsymbol{B}}, \mathcal{I}_{\boldsymbol{A}})\right]$$

where $Z(\mathcal{I}_{\boldsymbol{A}} \mid \boldsymbol{w}) = \sum_{\mathcal{I}_{\boldsymbol{B}}} \tilde{P}_{\boldsymbol{w}}(\mathcal{I}_{\boldsymbol{B}}, \mathcal{I}_{\boldsymbol{A}})$, n_i is the number of true groundings of the i-th rule or clause involving query atoms, and $R_{\mathcal{I}_{\boldsymbol{B}}}$ is the set of rules with at least one grounding involving a query atom.

In this formula the summation is restricted to formulas involving query atoms, and the partition function only involves summing over the different interpretations $\mathcal{I}_{\boldsymbol{B}}$, which are the settings over the query atoms. In this setting, we do not try to *generate* the entire distribution, but only the conditional distribution given values to the evidence atoms.

As we have discussed, learning Markov Logic Networks can be done by computing all groundings, and learning on the ground Markov Networks. Learning a Markov Network in this discriminative setting means to train the Markov Network as a *conditional random field* (CRF). Maximizing the log-conditional-likelihood involves maximizing $\sum_{j=1}^{k} \ln P(\mathcal{I}_{\boldsymbol{B}}[j] \mid \mathcal{I}_{\boldsymbol{A}}[j], \boldsymbol{w})$ which involves many different log-likelihood functions, corresponding to the different observations. Optimizing this objective is done by the same methods as above, and involves using the gradient of the log-conditional-likelihood:

$$\frac{\partial}{\partial w_i} l_{\mathcal{I}_{\boldsymbol{B}}|\mathcal{I}_{\boldsymbol{A}}}(\boldsymbol{w} : \mathcal{D}) = n_i(\mathcal{I}_{\boldsymbol{A}}, \mathcal{I}_{\boldsymbol{B}}) - \sum_{\mathcal{J}_{\boldsymbol{B}} \in SM[\Pi]} P_{\boldsymbol{w}}(\mathcal{J} \mid \mathcal{I}_{\boldsymbol{A}})(n_i(\mathcal{I}_{\boldsymbol{A}}, \mathcal{J}_{\boldsymbol{B}})) \tag{3.8}$$

for a single dataset, where we have used $n_i(\mathcal{I}_{\boldsymbol{A}}, \mathcal{I}_{\boldsymbol{B}})$ to emphasize that the counts are computed over the combined data. In words, the difference lies in computing the expected feature counts given $\boldsymbol{w}$, which is done *given the values of the evidence variables* $\boldsymbol{A}$. The second term computes (given $\boldsymbol{w}$) the counts of all stable models over the query variables, given the evidence variables.

Unlike performing gradient ascent over Eq. (3.5), where at every step of the optimization procedure we only need to compute these expected counts *once*, in the conditional case we need to compute this for every data instance, as this is conditioned on, at every iteration(Koller and Friedman 2009, p. 951). An advantage of this objective however, is that all the inference passes are usually easier to perform, because the size of the model is reduced and consists only of the query variables. This is especially the case if the evidence variables can take on many different values: having these fixed in the learning scenario might reduce the size of the Network significantly, cutting down the computational cost drastically.

The upshot is that conditional likelihood can be more advantageous to optimize, if the domain of the evidence variables is very large. In this case the size of the models would be reduced significantly by conditioning on these, making for faster optimization despite the higher number of required inference passes.

3.1.2 Bias and Overfitting

The structure of the Network in terms of the query and evidence variables is not the only thing that indicates which of the two general methods described will perform best on a specific dataset. One of the most important factors is the *bias* of the model: the assumptions a learning method makes on the form of a distribution. A generative model can be decomposed as follows:

$$P_{\Pi}(\mathcal{I}_{\boldsymbol{B}}, \mathcal{I}_{\boldsymbol{A}}) = P_{\Pi}(\mathcal{I}_{\boldsymbol{B}} \mid \mathcal{I}_{\boldsymbol{A}}) \cdot P_{\Pi}(\mathcal{I}_{\boldsymbol{A}}).$$

From this we can easily see that generative learning involves getting a good fit on the conditional likelihood of $\boldsymbol{B}$ given $\boldsymbol{A}$, in addition to getting a good model for $P_{\Pi}(\mathcal{I}_{\boldsymbol{A}})$. This second part involves computing the likelihood of the (assumed independent) evidence variables $\boldsymbol{A}$ or $\mathcal{I}_{\boldsymbol{A}}$; discriminative learning does not do this.

This extra objective inherent in generative learning constrains the model further in terms of assumptions on the probability distribution. Doing so can have the effect of providing *regularization* on the learned model, which can help overfitting on the data. This is because generative learning methods have more bias than discriminative learning methods. This can be especially helpful in the case of small training data, where discriminative methods have the risk of overfitting on the data.

For learning tasks with more data, the bias is less useful for learning and can start to dominate the error of a learned model (Koller and Friedman 2009, pp. 709–711). Here discriminative learning is usually a better choice; it can also perform quicker than generative learning depending on the structure of the LP^{MLN}-program.

3.1.3 Choice of Algorithm for Optimization

Having chosen an objective to optimize, one still has to choose a particular optimization algorithm that optimizes the weights with respect to this objective. As shown in (Lowd and Domingos 2007), the particular optimization algorithm can have huge effects on the effectiveness of weight learning. Discussing all possible ways to perform (gradient based) optimization in this setting is beyond the scope of this thesis; we will here limit ourselves to a few that are useful in practice.

Standard gradient ascent requires stipulating the step size, which is a hard parameter to get right: the algorithm can move in the wrong direction easily. In addition it can converge very slowly, even if moving in the right direction, due to overshooting the optimum (too big a step size) or going too slowly in the direction of the optimum. Hence it can be beneficial to use, or approximate, the second-order partial derivatives with respect to feature functions (the Hessian). We will here list some possible gradient ascent methods which have been used and studied for Markov Logic Networks (Lowd and Domingos 2007):

- **Line search**, which, at every iteration, determines the direction in which to go in addition to determining the step size. It normally does this (depending on the

specific algorithm) by computing the function at specific points in the direction of the gradient, to take the maximum as the optimal step size at that iteration.

- **Conjugate Gradient Ascent**, which stipulates at every iteration that the gradient along an already taken direction remains zero. This avoids "zig-zag" behaviour of normal gradient ascent, which can make convergence much faster to attain. This method can be made efficient for MLN's by using the Hessian matrix to choose a step size (called *scaled* conjugate gradient descent).
- **Diagonal Newton** which uses the diagonalized Hessian multiplied with the gradient to determine the next step. It approximates the objective function locally with a quadratic function, and determines the optimal value in that local area using an approximation of the Hessian matrix.

In LP^{MLN} and MLN, the Hessian is the negative covariance matrix. In general, computing this matrix is infeasible, and for larger datasets or larger programs this has to be approximated using a sampling algorithm (Section 3.2.3). In addition, there is the possibility to determine step sizes *per weight*, which helps when the learning rates for the different weights differ significantly (making standard gradient descent extremely slow and less effective).

3.2 Approximate Methods

As mentioned, for LP^{MLN}, Markov Logic, and even Markov Networks, computing $Z(\boldsymbol{w})$ can be too costly to compute, making it necessary to have to use approximative methods to learn the weights. Here we can either use a different, approximate, learning objective, or approximate the original objective by using approximative inference methods at every iteration. We will consider the two options in turn.

3.2.1 Pseudo-log-likelihood Objective

One approach, which was also initially used for weight learning in Markov Logic Networks in (Richardson and Domingos 2006), is to maximize the *pseudo-log-likelihood*, denoted L_{PL}. Here we replace the likelihood objective with a more tractable variant, restricting for convenience to the case of one database (m=1):

$$L_{PL}(\boldsymbol{w} : \mathcal{D}) = \prod_k P(\mathcal{I}_k | \mathcal{I}_{-k}, \boldsymbol{w}) \tag{3.9}$$

where $\mathcal{I}_{-k} = \{\mathcal{I}_1, ..., \mathcal{I}_{k-1}, \mathcal{I}_{k+1}, ..., \mathcal{I}_z\}$ for the z atoms in the data. That is, we calculate the conditional probability of every k-th atom on the values of all the other atoms. Because of the structure of Markov Networks, this is equal to

$$L_{PL}(\boldsymbol{w} : \mathcal{D}) = \prod_k P(\mathcal{I}_k | \mathcal{N}(\mathcal{I}_k), \boldsymbol{w}), \tag{3.10}$$

that is, conditioning only on the Markov blanket of (the node corresponding to) $\mathcal{I}_k$. Calculating this objective is much easier, as follows from the following equation:

$$P(\mathcal{I}_k|\mathcal{N}(\mathcal{I}_k)) = \frac{P(\mathcal{I}_k, \mathcal{N}(\mathcal{I}_k))}{P(\mathcal{N}(\mathcal{I}_k))} = \frac{\tilde{P}(\mathcal{I}_k, \mathcal{N}(\mathcal{I}_k))}{\sum_{\mathcal{I}'_k} \tilde{P}(\mathcal{I}'_k, \mathcal{N}(\mathcal{I}_k))}$$

Here $\tilde{P}$ is the unnormalized measure, and thus the global partition function $Z(\boldsymbol{w})$ has canceled out, and instead we only have to sum over the possible values of $\mathcal{I}_k$, denoted $\mathcal{I}'_k$. In the case of Markov Networks and LP$^{\text{MLN}}$, this consists of considering only both values of all the atoms in $\mathcal{I}$ conditioned on atoms that occur in the same formulas (all of its neighbours), which can generally be performed very efficiently.

The gradient of the pseudo-log-likelihood for Markov Logic Networks is:

$$\begin{aligned}\frac{\partial}{\partial w_i} l_{PL}(\boldsymbol{w} : \mathcal{D}) = \sum_{k=1}^{z} n_i(\mathcal{I}) - P_{\boldsymbol{w}}(\mathcal{I}_k = 0|\mathcal{N}(\mathcal{I}_k)) \cdot n_i(\mathcal{I}_{\mathcal{I}_k=0}) \\ - P_{\Pi_{\boldsymbol{w}}}(\mathcal{I}_k = 1|\mathcal{N}(\mathcal{I}_k)) \cdot n_i(\mathcal{I}_{\mathcal{I}_k=1})\end{aligned} \tag{3.11}$$

where $n_i(\mathcal{I}_{\mathcal{I}_k=1})$ is the number of true groundings of the i-th formula in $\mathcal{I}$, given that the k-th atom is set to 1 (or 0). This does not require inference over the model, and therefore optimizing l_{PL} is very efficient.

The important question with this approximate objective is of course how well it approximates the true log-likelihood objective. Under a very specific scenario, namely where the size of the training set approaches infinity, the probability that the parameter $\hat{\boldsymbol{w}}$, according to which the data is distributed, is a global optimum of the pseudo-likelihood objective, approaches 1 (Koller and Friedman 2009, p. 972). Thus, as m approaches infinity, the maximum pseudo-(log)-likelihood $\tilde{\boldsymbol{w}}_{PL}$ is almost certainly equal to $\hat{\boldsymbol{w}}$. So under this (theoretical) scenario, it is a sound approximation.

The problem here is the (impossible) assumption. It can be understood loosely as consisting of two criteria. First, the rules of the LP$^{\text{MLN}}$ program—here the model being learned—needs to be expressive enough so that it can represent the generating distribution; second, the training data needs to be large enough to properly express this generating distrbution. This second criteria can only be ensured at the infinitely large sample limit, and although it is likely to hold for sufficiently largre datasets, it is frequently problematic in practice due to too small training sets. In addition, if the model is not sufficiently expressive, training with respect to the log-likelihood and the pseudo-log-likelihood can give very different results.

In practice, pseudo-likelihood learned models can perform very well. However, in the case of dependencies between variables that are not neighbors of one another in the graph corresponding to the Network (Section 2.2.1) —long-range dependencies—pseudo-likelihood based models can fail to capture the dependencies properly. This is the case if the data does not contain certain values of neighboring variables; the assumption is that all values of the neighboring variables are fully observed and present in the training

data. This can be seen by considering that it conditions on values of the neighboring variables; if not all values of these are fully observed, dependencies with other variables can be lost entirely in the PLL-model (Koller and Friedman 2009, p. 973). The quality of pseudo-likelihood based models therefore depends very much on the model and data.

3.2.2 Contrastive Divergence

Another possible approximate objective can best be motivated by considering both the likelihood and pseudo-likelihood learning method as attempting to contrast the actually observed data (the training set) with other possible interpretations. This can be seen by considering Eq. (3.3), where we try to increase the distance between the expectation over the data, and the sum of all other instances or interpretations. The pseudo-log-likelihood attempts to make this easier, by contrasting the dataset with log-measures of interpretations where precisely one variable is flipped, as can be seen by considering Eq. (3.11).

The problem with the (contrastive) task in normal likelihood learning, is that the second term is very expensive to compute. The hope of a method called *contrastive divergence* is that we can provide other interpretations or instances relatively quickly, which still allow to move the learning procedure in the right direction. Here it has been found that using only a small number of MCMC-samples frequently already indicates the right direction for gradient ascent (Koller and Friedman 2009, p. 975; Lowd and Domingos 2007, p. 4), and provides enough of a contrastive term. This allows the gradient search to be much quicker than running the sampling algorithm until it approximates the posterior properly (which it does in the limit as the number of samples goes to infinity).

3.2.3 Learning by approximate inference

Alternatively, one can try to *approximately* optimize the exact likelihood objective, using an approximate inference procedure. There are multiple methods available to perform approximate inference in Markov (Logic) Networks.[5] A popular approach to perform approximate inference in Markov Networks is to generate instantiations of a Network, which are assignments to a subset of the random variables in a Network. This is a Monte Carlo approach, where instantiations of the random variables are drawn with the hope that they approximate the posterior distribution of the Network in reasonable time.

A particularly useful and popular class of sampling methods for Markov Networks are *Markov Chain Monte-Carlo* (MCMC) sampling methods, where a sequence of samples are taken from the data that provably get closer and closer to the posterior distribution. A popular example of a MCMC-sampling algorithm is the Gibbs-sampling algorithm. Generally, MCMC-methods work by using a *Transition model* $\mathcal{T}$ to create samples, that specifies the probability $\mathcal{T}(\mathcal{I} \to \mathcal{I}')$ of going from state $\mathcal{I}$ to $\mathcal{I}'$ in one step. If this

[5] See again (Koller and Friedman 2009); examples are different sampling methods, belief propogation, and variable elimination. We will here limit ourselves to the sampling methods which have been used for weight learning in either LP^{MLN} or Markov Logic.

transition model is properly chosen or designed, it is the case that the sampling algorithm gets closer to the posterior distribution. For example, in Gibbs-sampling, this is done by stipulating the transition probability for some atom to be the probability of that atom, given the values of all other atoms.

The difficulty for using MCMC methods with success lies in designing this transition model. Briefly, the Markov chain has to be *regular* and satisfy the *detailed balance* equation relative to the posterior distribution P_{Π}. Here the former means that for any k number of steps, and any two interpretations, there is a non-zero probability that one of the interpretations can be reached from the other in exactly k steps; detailed balance asserts that, given P_{Π}, $P_{\Pi}(\mathcal{I})\mathcal{T}(\mathcal{I} \to \mathcal{I}') = P_{\Pi}(\mathcal{I}')\mathcal{T}(\mathcal{I}' \to \mathcal{I})$. In LP$^{\text{MLN}}$ and Markov Logic, because of the stable model-based semantics and the first-order based semantics, deterministic dependencies are frequently present. In case of deterministic dependencies or just strong correlations, convergence time can be very slow or can even fail entirely for classical sampling methods such as Gibbs sampling (Koller and Friedman 2009, p. 515; Poon and Domingos 2006).

Because of this the specific MC-SAT algorithm has been created as an efficient approximate inference method that is able to handle such deterministic dependencies between variables. Recall that hard rules in LP$^{\text{MLN}}$ are encoded with the weight α, denoting the infinite weight. Optimizing the weights of rules in LP$^{\text{MLN}}$ is therefore analogous to weighted-$MAXSAT$ over Boolean formulas. As already discussed, inference in LP$^{\text{MLN}}$ is at least as hard as first-order logic, and because it involves maximization of rules, at least $FP^{\mathbf{NP}}$-hard. Although weighted-$MAXSAT$ is an untractable problem (finding the maximum weight that can be simultaneously satisfied is in $FP^{\mathbf{NP}}$), efficient heuristics and approximation methods exist for SAT related problems, which can usually find solutions very quickly: in particular, the MaxWalkSAT algorithm (Kautz, Selman, and Jiang 1996) for the weighted-$MAXSAT$ optimization problem.

This MaxWalkSAT algorithm has been combined with simulated annealing to produce an algorithm that near-uniformly samples satisfying assignments for a set of clauses in reasonable time (Wei, Erenrich, and Selman 2004). Simulated annealing is very good for sampling uniformly, but very slow for finding the assignments, while MaxWalkSAT does not sample uniformly; the combination provides a near-uniform and efficient sampling method. This (near-) uniform sampling is required for sampling in Markov Networks, and so the combination of MaxWalkSAT and simulated annealing therefore also allows for an efficient sampling procedure in Markov Logic: Poon and Domingos have implemented this in the MC-SAT algorithm (Poon and Domingos 2006).

The MC-SAT algorithm is described in pseudo-code below. In the algorithm, M is a subset of currently satisfied clauses that must also be satisfied in the next step of the algorithm. At every iteration, a state is randomly sampled from $\mathcal{U}_{SAT(M)}$, the uniform distribution over the set $SAT(M)$ (the set of states that satisfy M). At the next iteration, clauses satisfied by our taken sample can be, with probability $1 - e^{-w_k}$, added to M, which continues for the specified number of iterations.

MC-SAT Algorithm
$\mathcal{I}^(0) \leftarrow$ Satisfy(hard clauses)
for $i = 1$ to *numsamples* **do**
  M $\leftarrow \emptyset$
  for all $c_k \in$clauses satisfied by $\mathcal{I}^{(i-1)}$ **do**
    With probability $1 - e^{-w_k}$ add c_k to M
  end for
  Sample $\mathcal{I}^{(i)} \sim \mathcal{U}_{SAT(M)}$
end for

This MC-SAT algorithm has been adapted to work for LP^{MLN} (Lee and Y. Wang 2018). It adapts the MC-SAT algorithm slightly to accord with the penalty formulation (Section 2.3.3), so that ground instances of rules that are *false* in $\mathcal{I}^{(j-1)}$ are added to M with probability $1 - e^{w_k}$, and probabilistic stable models of $\mathcal{I}^{(j)}$ that satisfy *no* rules in M are chosen at every iteration.

Both MC-SAT (for MLN) and MC-ASP (for LP^{MLN}) are regular and satisfy the detailed balance with respect to $P_{\mathbb{L}}$ Eq. (2.3) resp. P_{Π} (Lee and Y. Wang 2018). This sampling method therefore works even for (near) deterministic dependencies, and is therefore a useful sampling algorithm for these formalisms.

MAP-Based Learning

An alternative inference method which approximates the expected counts $E_{\boldsymbol{w}}(n_i)$ of all rules or formulas, which constitutes the partition function, is to compute the counts in the single Maximum A-Posteriori assignment (MAP-assignment) (Koller and Friedman 2009, p. 967). The Maximum-a-Posteriori query, also called the Most Probable Explanation of the distribution, is the most probable assignment to all the (non-evidence) random variables, and is generally much easier to compute. Thus, given a parameter setting $\boldsymbol{w}$, non-evidence (query) atoms $\boldsymbol{B}$ and (possibly empty) evidence atoms $\boldsymbol{A}$, the aim here is to find:

$$MAP(\Im_{\boldsymbol{B}} \mid \mathcal{I}_{\boldsymbol{A}}) = argmax_{\mathcal{I}_{\boldsymbol{B}}} P_{\boldsymbol{w}}(\mathcal{I}_{\boldsymbol{B}}, \mathcal{I}_{\boldsymbol{A}})$$

where $\Im_{\boldsymbol{B}}$ is the set of all interpretations over the query atoms. This means that we are looking for the most probable joint assignment given the current weight settings $\boldsymbol{w}$.

To use this in learning, rather than contrast the counts of our data with the expected counts, we contrast it with the MAP-assignment, which is usually easier to compute. To calculate the gradient at every iterations (given a parameter setting $\boldsymbol{w}$ of that iteration), we compute:

$$\boldsymbol{E}_{\mathcal{D}}[n_i] - n_i[\mathcal{I}^{MAP}(\boldsymbol{w})]$$

where $\mathcal{I}^{MAP}(\boldsymbol{w})$ is the MAP-interpretation (assignment).

Using this gradient in learning means we effectively try to optimize the following approximate objective (for a single data instance):

$$l(\boldsymbol{w} : \mathcal{D}) - \ln P(\mathcal{I}^{MAP}(\boldsymbol{w}) \mid \boldsymbol{w}). \tag{3.12}$$

In optimizing for this objective, the partition function does not need to be computed, because it cancels out over the two terms.

However, this objective has noticeable problems: Eq. (3.12) has its maximum either when the counts over the data match the counts in the MAP-setting, *or* when all weight settings $\boldsymbol{w}$ are set to zero (Koller and Friedman 2009, pp. 967–968). Furthermore, for Markov Logic and LP$^{\text{MLN}}$, the MAP-assignment might not be unique.

However, it was successfully used in (Singla and Domingos 2005) using MaxWalkSat, where they used the average over the weights to counteract the effect of overfitting on the dataset, at every iteration; furthermore, they initialized the weights at the log-odds of the weights with the goal of making learning faster.

Log-Odds calculation

As described in (Richardson and Domingos 2006), an intuitive understanding of a weight of a formula in an LP$^{\text{MLN}}$-program or a Markov Logic Network, is as the logarithm of the odds ratio, other things being equal. This follows immediately from Eq. (2.4), where the unnormalized weight corresponds to the exponent of the sum of weights of rules. Thus, the log-odds of a rule R in an interpretation $\mathcal{I}$, corresponds to:

$$\log\left(\frac{n_R}{|\{r : r \in gr(R), \mathcal{I} \not\models\} r|}\right). \tag{3.13}$$

Here n_R refers to the number of times rule R is satisfied under $\mathcal{I}$ (i.e., number of true groundings), and the denominator counts the number of groundings of R which are violated under $\mathcal{I}$.

This interpretation of the weight of a rule is only correct if the rules are independent: in general, changing the truth value of one rule influences the number of times another rule is violated or satisfied. This is simply because variables are shared among rules; again, the edges between predicates in the graph corresponding to an LP$^{\text{MLN}}$ Network represent these dependencies.

In (Eiter and T. Kaminski 2016), the log-odds was the only used learning method, which was adopted as an approximation of the real distribution. This learning method only involves a single calculation per rule, and can be seen as assuming independence between constraints. This is generally a false assumption to make, but can sometimes work well; it performed well as a learning method in (Eiter and T. Kaminski 2016).

3.3 Other issues in learning

The previous sections have dealt with the theoretical or mathematical properties of weight learning, and how to efficiently perform weight learning with respect to the possible

objectives. In doing so we have stepped over some related issues that might occur, depending on the nature of the dataset.

3.3.1 Multiple interpretations

In many of the equations described above we have provided the equation for multiple interpretations or stable models provided as training data. Equation (3.7) can easily be extended to learn over multiple data instances (stable models $\mathcal{I}_1, ..., \mathcal{I}_m$) in the training set:

$$\frac{\partial \ln P_{\Pi_{\boldsymbol{w}}}(\mathcal{I}_1, ..., \mathcal{I}_m)}{\partial \boldsymbol{w}_i} = \sum_{j \in \{1...m\}} \left(-n_i(\mathcal{I}_j) + \sum_{\mathcal{J} \in SM[\Pi]} P_{\Pi_{\boldsymbol{w}}}(\mathcal{J}) \cdot n_i(\mathcal{J}) \right). \tag{3.14}$$

The only difference then lies in computing the counts n_i for every iteration, as the second term of the summation is the same.

For the conditional likelihood (as in Eq. (3.8)), this is slightly different, where the following equation describes the gradient over multiple stable models:

$$\frac{\partial \ln P_{\Pi_{\boldsymbol{w}}}(\mathcal{I}_1, ..., \mathcal{I}_m)}{\partial \boldsymbol{w}_i} = \sum_{j \in \{1...m\}} \left(-n_i(\mathcal{I}_{\boldsymbol{A},j}, \mathcal{I}_{\boldsymbol{B},j}) + \sum_{\mathcal{J}_{\mathcal{B}} \in SM[\Pi]} P_{\Pi_{\boldsymbol{w}}}(\mathcal{J}_{\mathcal{B}} \mid \mathcal{I}_{\boldsymbol{A},j}) \cdot n_i(\mathcal{J}, \mathcal{I}_{\boldsymbol{A},j}) \right). \tag{3.15}$$

where the right term involves summing over the probabilities of all stable models $\mathcal{J}_{\boldsymbol{B}}$ involving a query predicate, *given* the values to atoms $\boldsymbol{A}$ (groundings of evidence predicates) as given by the j-th stable model ($1 \leq j \leq m$).

Alternatively, learning from multiple stable models can be reduced to learning from a single stable model, as shown in in (Lee and Y. Wang 2018). Lee and Y. Wang describe a method that does this by introducing an extra argument to every predicate, indexing the stable model; an alternative method, used in Chapter 4, is to encode the dataset in every element of the domain (e.g. "Alice-j","Alice-k"). Intuitively, this makes all the datasets disjoint and separates all information, so that the groundings of rules over different datasets are distinct.[6] By exactly the same reasoning as Theorem 3 of (Lee and Y. Wang 2018, p. 15), combining datasets is then equivalent in terms of the probability distribution, to learning on the separate datasets and using the product of the probabilities.

3.3.2 Ill-conditioned data

For general gradient-descent algorithms, especially those using a simple update procedure as standard gradient-descent, ill-conditioning of the data is frequently an issue. Ill-conditioned data means, roughly, that a small difference in the output variable can have a very big effect on the variable to be optimized. If the *condition number*—a measure

[6] The groundings of any two ground programs, obtained by two datasets with such an encoding in their constant-names, is then independently divisible so that their programs are independent (B. Wang et al. 2018).

on the effect a change has in input variables, on the output variables—is very large, the learning problem is said to be ill-conditioned, otherwise well-conditioned.

In general, the condition number is given by:

$$\lim_{\epsilon \to \infty} \sup_{\|\delta_x\| \leq \epsilon} \frac{\|\delta f\|}{\|\delta x\|}$$

where $\|\cdot\|$ is a norm function, for example euclidean norm. In the specific case of our learning problem using gradient descent, the condition number is the ratio of the largest and smallest eigenvalues of the Hessian matrix.

In ill-conditioned problems, because of the corresponding large difference between counts of rules, no learning weight is appropriate for all weights, which can significantly decrease or slow down the effectiveness of gradient descent. Per-weight learning rates are reported by Lowd and Domingos as being especially successful if some rules (clauses, formulas) have significantly higher counts (or violations, depending on the implementation) than other rules, so that the learning rate is dominated by those former rules.

3.3.3 Missing data

In normal scenarios, and for all learning methods explained above, it is assumed that all data is complete: for any atom the data says whether it is true or false. In many cases, this does not hold of the data: not all values are witnessed and so the truth-value of some atoms is unknown from the data. This scenario of missing data, or incomplete interpretations, brings with it a number of new issues.

Unlike the case of fully observed data (investigated in the rest of this chapter), in the case of missing data we cannot simply maximize the likelihood of the observed data. This is because the distribution underlying the observed data decomposes into two, possibly connected, distributions: the distribution underlying the random variables (in our case, relations), and the *observation mechanism*. We are here not interested in learning the distribution of only the observed relations, but of all occurring values, including the missing ones.

A simple case of when the observation mechanism is of importance in learning weights, is when certain values of a random variable (atoms) are purposely deleted. As an example, we might do a simple coin-flip a number of times, except that the experimenter who gives us the values does not like "heads" (does not like the picture displayed) and so frequently will not note these values.(Koller and Friedman 2009, p. 849) Our observed data would then be heavily biased towards tails, which would not adequately represent the true distribution. However, if we sometimes miss data because the coin landed on the floor — assuming a perfect coin, or one where landing on the floor is independent on the value — our observed data would adequately represent the true distribution.

The theory of learning in the case of missing data is involved and we will here only be able to discuss certain aspects. Generally, if there is missing data, the data is generated first

according to the data distribution or model, and secondly according to the observability model (Koller and Friedman 2009, p. 851). This means that there are, in general, two sets of parameters that have to be learned to understand the process that generated the dataset: in addition to the normal weights, we have the parameters that define the observability model. Since these could be combined (as in the first coin-toss example), learning the real model can be highly difficult in the case of missing data, even if we are only interested in the model that describes the data distribution.

Of particular interest therefore are situations with missing data satisfying some properties. For this, we will phrase the theory in terms of general random variables: in the case of LP^{MLN}, these random variables are the truth values of relations. Define for any random variable X the observability variable O_X, the value of which (written o_X) says whether the value of X is observed. The first is as the second coin-toss example, where whether a value is observed or not is completely random: this is called *Missing Completely At Random* data (MCAR). More formally:

Definition 3. A missing data model $P_{missing}$ is MCAR if $P_{missing} \models (X \perp O_X)$ for all random variables X.

where $X \perp Y$ means (as is common) that X and Y are independent events. This means that the observation mechanism is independent of the distribution underlying X: whether we observe an atom or not is independent of its value.

The second situation is when data is said to be *Missing At Random* (MAR). To introduce this, let z be a tuple of observations: values of (some of) the random variables. We partition the random variables $\boldsymbol{X}$, into observed $\boldsymbol{X}^z_{obs}$ and hidden $\boldsymbol{X}^z_{hidden}$ ones, such that the value of X_i in the observed set $\boldsymbol{X}^z_{obs}$ is known in an observation z, and unknown in $\boldsymbol{X}^z_{hidden}$. Then we can define MAR as follows:

Definition 4. (Koller and Friedman 2009, p. 854) A model $P_{missing}$ is MAR if, for all observations z with non-zero probability and for all possible values of the hidden random variables, $\boldsymbol{x}^z_{hidden}$, we have $P_{missing} \models (o_{\boldsymbol{X}} \perp \boldsymbol{x}^z_{hidden} \mid \boldsymbol{x}^z_{obs})$.

In words, *given* the values in an observation z of observed variables $\boldsymbol{x}^z_{obs}$, the values of the observability variables $O_{\boldsymbol{X}}$ of $\boldsymbol{X}$ are independent of $\boldsymbol{x}^z_{hidden}$. This is a somewhat contrived definition, but it says, intuitively, that the values of the observed variables fully account for the hidden variables: the observability mechanism will not provide new information on the hidden variables. Note that MCAR $\subset$ MAR (taking both as sets of models).

By contrast, data is *not* missing at random when the value of a variable that is missing, is related to the reason it is missing.

MCAR and MAR are important situations, because the likelihood function of the missing data can be written as the product of two likelihood functions: one for the observability model, and one for the data model (Koller and Friedman 2009, p. 855). Thus, we can

ignore the observability model (which is generally unknown) in optimizing the weights; this is generally not the case for situations that are not M(C)AR.

Learning on missing data

Even though the optimal weights can be optimized using standard gradient ascent in the case of data missing at random, there are normally multiple interpretations or stable models that satisfy the partial evidence.

In this case, if we consider the data as a single ground formula F, the probability of the data which is used in the optimization algorithm with respect to LP^{MLN} program Π (Lee and Y. Wang 2018, p. 5), is:

$$P_\Pi(F) = \frac{\sum_{\mathcal{I} \models F, \mathcal{I} \in SM[\Pi]} W_\Pi(\mathcal{I})}{\sum_{J \in SM[(]\Pi)} W_\Pi(J)}. \tag{3.16}$$

Here there are multiple further possibilities to optimize the likelihood function (Koller and Friedman 2009). For our experiments in Chapter 4, the Expectation Maximization (EM) algorithm (Bishop 2006) is used. This attempts to fill in, or complete, the missing data optimally using the current parameter settings at every iteration.

3.3.4 Lifted Learning

Before turning to a learning example, the method of *lifted* learning methods deserves attention. LP^{MLN} and MLN are first-order logic based formalisms, but all methods described in this chapter involve computing all the groundings, and learning weights on the grounded Markov Network. Computing all groundings can create an immense number of random variables, making learning and inference very complicated. Lately, efforts have been made to perform inference and learning on lifted MLN's, which use symmetries in the structure of the lifted (first-order) model to significantly speed up inference (Kimmig, Mihalkova, and Getoor 2015; Van Haaren et al. 2016). These efforts have shown lifted inference and learning methods can be very efficient, while also giving very accurate results. However, because of the extra time needed to study these topics in detail and because we have not been able to get these methods to work in Chapter 4, we will not pursue these methods and implementations further in this work.

3.4 Learning example – Virus dataset

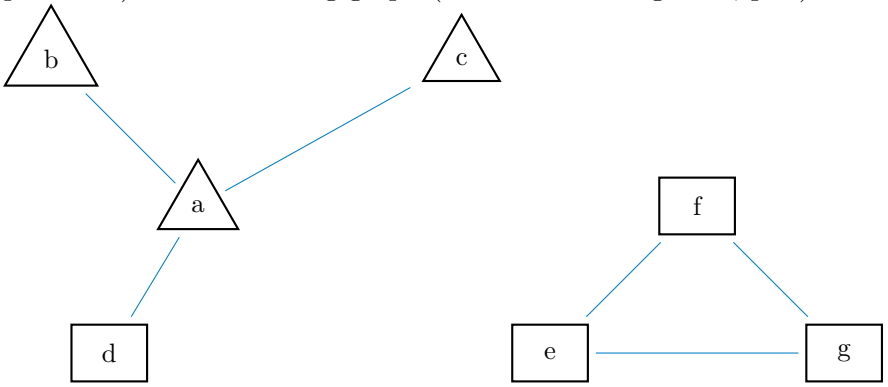
To illustrate some of the concepts and methods illustrated in this chapter, we will here consider a simple dataset and train the weights on this dataset. We use the artificial toy "Virus" dataset from (Lee and Y. Wang 2018), which serves to illustrate some of the aspects of learning and of the LP^{MLN} formalism. The goal is to predict the probability that someone carries a virus, given certain relations between people. This is a collective classification problem, where a good prediction should take into account the relations between the people in the domain.

The LP^{MLN} program under consideration consists only of the following two rules:

Program 3.1

$$Hasdisease(x) \leftarrow Carriesvirus(x) \quad : w$$
$$Carriesvirus(y) \leftarrow Contact(x, y), Carriesvirus(x) \quad : w$$

A version of the dataset is available at (*LPMLN learning system* 2019), although we will use the adapted version described in (Lee and Y. Wang 2018, p. 6). This adapted version can be represented (with respect to the "Contact" relation and the "Carriesvirus" predicate) in the following graph (Lee and Y. Wang 2018, p. 6):



Where the edges represent the contact relation, triangles represent people that carry the virus, and rectangles represent people that do not carry the virus. The edges are undirected since the "Contact" relation is symmetric. We state that "a" and "b" have the disease.

Using the LP^{MLN}-learn system, learned weights differ rather strongly between algorithms. See table. Pseudo-log-likelihood is clearly the fastest with only 5 seconds, while MC-ASP takes 15 minutes and Gibbs sampling takes 10 minutes. Metropolis-Hastings sampling, theoretically better, also performs much faster than Gibbs with one-and-a-half minutes. For MC-ASP we used 100 max iterations with 100 samples each iteration, and a learning rate of 0.1. Results did not change significantly when changing these parameter settings. Noticeable in the results is that different weights do not significantly alter the predictions, and all manage to capture the structure of the Network and the influence that has on the "Carriesvirus" predicate.

Using Alchemy, MCSAT learning with 100 iterations and max 100 MCSAT samples at each iteration, learning is extremely fast at under a second. Letting Alchemy choose the size of the sample (it chooses around 10000) learning takes 1 minute and 5 seconds. This makes it significantly faster than the LP^{MLN} learning algorithm. This is partly due to the implementation, but presumably also since the LP^{MLN}-learn system does not distinguish in query and evidence predicates so does not clearly perform discrimative learning, as Alchemy does. Indeed, training only on the "Carriesvirus" predicate in Alchemy makes learning slower by half a minute, because there are more predicates to be conditioned on.

Table 3.1: LP^{MLN}-learn virus dataset

	PLL	**Gibbs**	**MC-ASP**
Weights	Rule 1: 0.67 Rule 2: 0.65	Rule 1: 0.69 Rule 2: 0.28	Rule 1: 0.923 Rule 2: 0.233
Predictions	Carriesvirus(c) 0.74 Carriesvirus(d) 0.74 Carriesvirus(a) 1.0 Carriesvirus(b) 0.74 Hasdisease(c) 0.49 Hasdisease(d) 0.49 Hasdisease(a) 0.66 Hasdisease(b) 0.49	Carriesvirus(c) 0.66 Carriesvirus(d) 0.66 Carriesvirus(a) 1.0 Carriesvirus(b) 0.66 Hasdisease(c) 0.44 Hasdisease(d) 0.44 Hasdisease(a) 0.66 Hasdisease(b) 0.44	Carriesvirus(c) 0.63 Carriesvirus(d) 0.63 Carriesvirus(a) 1.0 Carriesvirus(b) 0.63 Hasdisease(c) 0.45 Hasdisease(d) 0.45 Hasdisease(a) 0.71 Hasdisease(b) 0.45

Table 3.2: Translated Alchemy learn virus dataset

	PLL	**MC-SAT**	**LPMLN-Learn**
Weights	Rule 1: -1.49073 Rule 2: 0	Rule 1: 0.42 Rule 2: -4	Rule 1: 0.923 Rule 2: 0.233
Predictions	Carriesvirus(a) 0.99 Carriesvirus(b) 0.99 Carriesvirus(c) 0.99 Carriesvirus(d) 0.99 Carriesvirus(e) 0.99 Carriesvirus(f) 0.99 Carriesvirus(g) 0.99 Hasdisease(a) 0.64 Hasdisease(b) 0.59 Hasdisease(c) 0.61 Hasdisease(d) 0.70 Hasdisease(e) 0.68 Hasdisease(f) 0.58 Hasdisease(g) 0.58	Carriesvirus(a) 0.99 Carriesvirus(b) 0.02 Carriesvirus(c) 0.01 Carriesvirus(d) 0.02 Carriesvirus(e) 4.99e-05 Carriesvirus(f) 0.99 Carriesvirus(g) 0.99 Hasdisease(a) 0.63 Hasdisease(b) 0.01 Hasdisease(c) 0.01 Hasdisease(d) 0.02 Hasdisease(e) 4.99e-05 Hasdisease(f) 0.60 Hasdisease(g) 0.64	Carriesvirus(a) 0.99 Carriesvirus(b) 0.61 Carriesvirus(c) 0.64 Carriesvirus(d) 0.58 Carriesvirus(e) 0.57 Carriesvirus(f) 0.58 Carriesvirus(g) 0.56 Hasdisease(a) 0.74 Hasdisease(b) 0.47 Hasdisease(c) 0.498 Hasdisease(d) 0.44 Hasdisease(e) 0.43 Hasdisease(f) 0.43 Hasdisease(g) 0.40

We have here tried also a different gradient ascent procedure, which gives exactly the same results as the standard gradient ascent.

The results of Alchemy based learning on the translated input are in Table 3.2. As already reported in (Lee and Y. Wang 2018, p. 7), learning under the MLN semantics fails to capture the relation between the persons in the domain. The structure given by the contact relation under this interpretation is not visible in the results at all. An explanation for this is that the LPMLN2MLN system only gives an equivalent Markov

Logic Network for absolutely tight programs as described in Section 2.3.2. Since the positive dependency graph of the virus program has cycles, the program resulting from the LPMLN2MLN translation is not equivalent. This also explains why stipulating the weights (using those learned with MC-ASP under LP^{MLN}-learn) does not give equivalent results, when using Alchemy for the inference. However, given the weights as calculated in LP^{MLN}-learn but using Alchemy for inference, the result is slightly better, as we can see in Table 3.2, third column.

This learning example on the virus dataset already illustrates that the different methods are not equivalent: the weights learned using different methods (PLL or MC-ASP sampling) are different. Also, we see here that the translation to MLN does not always work; this depends on the structure of the LP^{MLN} program. However, from this we do not have a good notion of accuracy of the learning methods. In the next chapter we will consider a learning problem where we can properly measure the quality of learned models.

CHAPTER 4

Experiments

4.1 Problem specification and LP$^{\mathrm{MLN}}$ program

In this chapter we perform multiple experiments related to weight learning in LP$^{\mathrm{MLN}}$. For all experiments, we use the image classification dataset from the *LabelMe* dataset (Russell et al. 2008), where LP$^{\mathrm{MLN}}$ was used by (Eiter and T. Kaminski 2016). This dataset contains 120 indoor and 120 outdoor images, which we shall split (following Eiter and T. Kaminski) in 30 training, 30 validation and 60 test images both. This makes it a relatively small dataset, which might have an impact on the effectiveness of learning methods.

Our experiments involve the following topics:

- Differences between learning methods and effectiveness on our dataset;
- Differences in parameter settings in learning methods;
- Relations between the rules and the corresponding weights;
- Learning on data involving missing data.

From Chapter 3 and while performing experiments, we obtained the following hypotheses that we consider in this chapter:

H1. MC-ASP/MC-SAT sampling methods outperform pseudo-likelihood based learning and the log-odds calculation.

H2. Different parameter settings influence prediction results of learning methods notably.

H3. The constraints of our dataset are mostly independent, such that learning the weights of the rules independently of one another is equally effective as combined learning.

H4. Using Expectation-Maximization (EM) outperforms standard MC-ASP/MC-SAT learning methods in the case of missing data, on identical parameter settings.

These hypotheses are tested with the goal of highlighting specific topics within weight learning in LP^{MLN}. In addition, general observations regarding weight learning on our dataset are surveyed. As a measure on the effectiveness of learned models, we test primarily on prediction accuracy. Prediction accuracy is measured here using the Jaccard similarity coefficient score J (Pedregosa et al. 2011). This compares the predicted labels with the true labels, according to the following formula:

$$J(y_{pred}, y_{true}) = \frac{|y_{pred} \cap y_{true}|}{|y_{pred} \cup y_{true}|} \tag{4.1}$$

In addition to this metric, we consider whether specific classes are represented in predictions, based on confusion matrices of predicted classes vs true classes.

The first hypothesis stems from results from experiments done on weight learning Markov Logic Networks (Singla and Domingos 2005; Lowd and Domingos 2007). In addition, the log-odds model is based on very strong assumptions on the distribution; we expected models with less assumptions to work better. The second hypothesis stems from (Lowd and Domingos 2007), where methods based on line search outperform standard gradient ascent methods. The third hypothesis arose from experiments done, where the log-odds computation performed very well in our tests; our hypothesis would explain this result. Finally, the fourth hypothesis arises from the theory of missing data scenarios, as explained in Section 3.3.3.

4.1.1 Data description

For both the indoor and outdoor set, we use the 12 labels that have been defined and used in (Eiter and T. Kaminski 2016):

- indoor: “chair” (c), “monitor” (mn), “keyboard” (k), “mouse” (ms), “table” (t), “book” (bk), “shelf” (s), “wall” (wl), “board” (br), “person” (p), “door” (d) and “window” (wi)
- outdoor: “sign” (sg), “person” (p), “tree” (tr), “window” (wi), “door” (d), “street” (st), “car” (c), “sky” (sk), “building” (b), “sidewalk” (si), “wheel” (wh) and “trunk” (trn).

.

The number of objects in a scene varies between 7 to 23 for the indoor, and 7 to 28 for the outdoor scenes. The goal of the classification is to assign the labels to (manually segmented) objects. In addition to these labels, the relations "contains", "close to", "above", "under", "overlaps", "contains in bottom part" and "higher". Using these relations, this becomes a collective classification problem, where the relations between objects should be taken into account for the classification.

The rules to be learned are all constraints, also as in (Eiter and T. Kaminski 2016). As an example rule of the encoding:

Program 4.1
$$\leftarrow assignedlabel(x,s), not\ containsBook(x) \quad : w.$$
$$containsBook(x1) \leftarrow contains(x1,x2), assignedlabel(x2,bk) \quad : \alpha.$$

Here s stands for "shelf"; the weak constraint intuitively says that shelves usually contain books.

In total there are 20 rules to be learned both for the indoor dataset and for the outdoor dataset. In (Eiter and T. Kaminski 2016) only the log odds were learned, assuming independence of the constraints:

$$w_i = \sum_{\mathcal{I} \in \mathfrak{I}_{KB}} \log\left(\frac{\text{numunsat}_i(\mathcal{I})}{\text{numsat}_i(\mathcal{I})}\right)$$

where numunsat_i is the number of times the i-th constraint is violated in $\mathcal{I}$, and numsat_i is the number of times it is satisfied. This method for learning the weights makes very strong assumptions on the distribution—independence of the constraints—but performed well in practice for the hybrid classification, almost consistently improving the result of the local classifier.

For the prediction, on the validation and test set, the "local" classifier (we use logistic regression as in (Eiter and T. Kaminski 2016)) is combined with the LP$^{\text{MLN}}$ program into a hybrid classifier. Here the LP$^{\text{MLN}}$ program that represents this hybrid encoding is given by Program 4.1 along with:

Program 4.2
$$a\_label(x,C) \leftarrow not\ not\_a\_label(x,C), object(x), label(C) \quad : \alpha.$$
$$not\_a\_label(x,C) \leftarrow not\ a\_label(x,C), object(x), label(C) \quad : \alpha.$$
$$\leftarrow 1 \neq \#count\{C : a\_labelProb(x,C,P)\}, object(x) \quad : \alpha.$$
$$\leftarrow not\ a\_labelProb(x,C,P), cl(x,C,P) \quad : P.$$
$$a\_labelProb(x,C,P) \leftarrow a\_label(x,C), cl(x,C,P) \quad : \alpha.$$

Here the $cl(x,C,P)$ is the probability P assigned by the local classifier to x belonging to C. *a_label* is short for *assigned_label*. Testing of the learned weights (both on validation and on the test set) is done by comparing with the non-hybrid classification model based only a local classifier as in (Eiter and T. Kaminski 2016).

For learning, it is only the weights in Program 4.1 that are to be learned, as the P-values in Program 4.2 are given by the local classifier. Our learning setup therefore separates the LP$^{\text{MLN}}$ program from the local classifier and does *not* involve Program 4.2. We include the following rules instead to create all possible interpretations, which constitute the learning LP$^{\text{MLN}}$ program together with the context constraints of form Program 4.1:

Program 4.3

$$not\_a\_label(x,C) \leftarrow not\, a\_label(x,C), object(x), label(C). \quad : w.$$
$$a\_label(x,C) \leftarrow not\, \neg a\_label(x,C), object(x), label(C). \quad : w.$$
$$\leftarrow 1! = \#count\{C : a\_label(x,C)\}, object(x) \quad : \alpha.$$

These rules are required to create all possible answer sets of the LP$^{\text{MLN}}$ program, to contrast the data against. All relational facts such as objects being above or left of another, or objects containing another, are given as facts in the input LP$^{\text{MLN}}$ program so that the answer sets differ only with respect to the rules (constraints) to be learned.

For the native LP$^{\text{MLN}}$-learn system, only one datafile can be given as input, so we encode the relevant dataset to every constants, e.g. "object1-000000". This is done as explained in Section 3.3.1, and equivalent in the learned weights to learning on every dataset separately (at each iteration).

Since all weights only relate to constraints in this setup, we perform inference using standard ASP with weak constraints directly, and no inference in LP$^{\text{MLN}}$ through a transformation to ASP is necessary. This can be seen by considering Section 2.3.3: the translation to ASP would include an atom *unsat* with weight equal to the constraint, which is a unnecessary step if all weighted rules are constraints. For learning, because of the differing semantics of the weights, the body B of every weak constraint is negated to give $\neg B$, and the weight of the learned LP$^{\text{MLN}}$ rule is flipped in sign. As $\neg B$ is only allowed syntax for clingo if B is a literal, we further translated this to equivalent rules in clingo syntax by introducing an auxiliary predicate *violated*. For the weak constraint of Program 4.1 we used:

Program 4.4

$$violated(i,x) \leftarrow assignedlabel(x,s), not\, containsBook(x) \quad : \alpha.$$
$$\leftarrow not\, violated(i,x) \quad : -w.$$

where i is the number of the constraint, and $-w$ indicates that the learned weight is the negative of the weight used in the hybrid-classification encoding. This translation can not be used for LPMLN2MLN learning, as here Alchemy memory usage quickly exploded

and within seconds Alchemy loaded 15 gb's in memory. However, in first-order logic the negative translation is easily encoded, since first-order formulas can easily be negated.[1]

Unlike in (Eiter and T. Kaminski 2016), we do not perform constraint selection, since this is not part of the weight-learning problem *per se*, but rather a part of learning the structure of the LP$^{\text{MLN}}$ program: it concerns learning the ideal combinations of rules, not learning the weights of those rules. In addition, the weight-learning method would ideally make weights of non-influential rules go to zero.

We also consider an interpretation of a rule as a Markov Logic formula, where a rule $A \leftarrow B$ is directly interpreted as the first-order formula $B \rightarrow A$. Here the weights are kept the same. This interpretation is not semantically equivalent — as explained in previous chapters, the LP$^{\text{MLN}}$ semantics extends the MLN semantics — and provides a simplified interpretation. However, this can still work or yield interesting practical results. We will refer to such an interpretation as a direct MLN interpretation.

We included the relational "facts" of every scene in the input program for the LP$^{\text{MLN}}$-learn except where mentioned otherwise. For example, "above(object1,object2)" and "contains(object2,object4)" are considered as facts. In this way, the learning method becomes discriminative, which makes sense in this setup since all such facts are given, and the prediction of labels of objects is queried.

4.2 Systems and setup

All tests were performed on a Dell XPS 9550 with Intel® Core$^{\text{TM}}$ i7-6700HQ CPU @ 2.60GHz 8 core processor running Ubuntu 18.10, although the learning methods are not setup for multiple core usage. It has 16 gb of RAM memory, and although the system has additional swap space, this turned out to be too slow for learning.

For native LP$^{\text{MLN}}$ learning the LP$^{\text{MLN}}$-learn system (*LPMLN learning system* 2019) is the only available system, featuring MC-ASP sampling-based learning algorithms, along with Pseudo-likelihood learning, Gibbs-sampling and MH-sampling. MC-ASP is a sampling algorithm based on MC-SAT, adapted for ASP. The methods based upon the MC-ASP are gradient ascent based learning methods, which differ in the sampling implementation used. The implementation using the near-uniform Xorro sampler (Everardo, R. Kaminski, and Lindauer 2019) is used in the "MC-ASP-EM" (expectation maximization) algorithm, while the standard MC-ASP uses an internal clingo "XOR-count" sampler. Internally, the LPMLNlearn system uses, LPMLN2ASP (Lee, Talsania, and Y. Wang 2019) to compute LP$^{\text{MLN}}$ using Clingo. The system uses both clingo versions 4 and 5: 5 for the MC-ASP methods, 4 for the other methods.

For the translation to Markov Logic, LPMLN2MLN is used, which is included in the LPMLN2ASP program. This allows using both Alchemy (*Alchemy: Open Source AI*

[1] Formulas cannot easily be negated in LPMLN2MLN input, so the negative translation is applied to the resulting MLN program.

2019) and Tuffy (Niu et al. 2011) for learning and inference. Alchemy-2 also allows lifted learning and extended inference mechanisms, although Alchemy version 1 already provides many learning algorithms: log-likelihood based generative learning, and voted perceptron, diagonalized Newton's method, and rescaled conjugate gradient ascent for discriminative learning. In addition, it allows Expectation Maximization to fill in missing data. In our experiments we frequently encountered problems using Alchemy 2 (thrown exceptions) which did not arise in the original version of Alchemy. Unless mentioned, we therefore used the original version 1 of Alchemy in our training.

We split the data into an equally sized training and test set, both containing 60 scenes (for both indoor and outdoor). We then subdivided the training set into equal training and validation sets; the validation set is meant for tuning the learning methods and to test (and possibly reject) hypotheses about settings.

The hybrid classification program Program 4.2 uses DLVHEX for the ASP (with weak constraints) computation, to input the results of the local classifier in the program. DLVHEX (Eiter, Mehuljic, et al. 2015) is a logic-programming reasoner used for HEX-programs, which are an extension of answer set programs which allow for external computation sources. In the hybrid classification program, this allows for injecting classifier weights and inferring atoms using Python libraries.

As the weights resulting from both LP$^{\text{MLN}}$-learn and Alchemy are decimal valued, and both Clingo and DLVHEX only support integer weights, we multiply the weights accordingly so that the resulting weights are integer-valued, yet capture most of the precision of the output weights of the learning methods. As an example, if this is not performed all weights between zero and a half would obtain the same weight when using clingo; essential precision would be lost. We generally scale to four decimal places (except when mentioned explicitly in the results).

For the accuracy scoring we use scikit-learn (Pedregosa et al. 2011). However, since these scores also involve the influence of the "local" (feature-based) classifier in the hybrid classification setup, these accuracy scores are not absolute measures of the effectiveness of the learning methods, but always relative to the performance of the local classifier. However, since this model is fixed in all tests, the performance of different learning methods can be properly assessed based on this score.

In order to be able to assess the effect of weight learning mostly independently of the local classifier, we also include in our tests an LP$^{\text{MLN}}$-program with all equal weights, for both the indoor and outdoor dataset. As explained in Section 2.2.2 and Section 2.3, this corresponds to looking for the interpretations that satisfy the maximum number of rules and can be interpreted as a "qualitative" weight-assignment. Including this provides a benchmark for any learning method.

4.3 General training

In this section we test the first and second hypotheses introduced in the beginning of this chapter. For this, we needed to test multiple models; we therefore first describe initial observations with respect to training, including the time it takes to learn models using specific methods, and then proceed to discuss general observations related to the accuracy performance of trained models.

4.3.1 Training time

Indoor Training times are listed in Table 4.1. The log-odds calculation as used in (Eiter and T. Kaminski 2016) calculates the weight in 12 seconds over the indoor dataset, but makes very strong independence assumptions on the weights and involves no optimization procedure.

Table 4.1: Training times, indoor dataset

	PLL-LPMLN	**MCASP-LPMLN**	**PLL-Alchemy**
Iterations	50	50 (50 samples)	336
Time	1 h	4 h 32 m	10 m
	MCSAT-Alchemy	**MAP-Alchemy**	**Log-Odds**
Iterations	100 (344 samples)	100 (344 samples)	n.a.
Time	6h. 28 m.	40 m.	12 sec.

Learning with the pseudo-likelihood objective for 50 iterations, training took 60 minutes. To determine the proper settings for gradient ascent, we trained several models over 20 iterations on only four scenes, with different parameter settings; learning these models takes only a few minutes to compute. For pseudo-likelihood learning with Alchemy after converting through LPMLN2MLN, 10 minutes and 17 seconds were required for the computation, consisting of 336 iterations of the L-BFGS algorithm. By far the biggest amount of the computing time lies in computing the counts, however, as the L-BFGS algorithms only takes a few seconds to complete all the iterations. Since the L-BFGS algorithm avoids the problems of standard gradient ascent, such as the starting point (which only influences convergence time, due to there being no local minima/maxima), step size, and direction, different models are not necessary to consider here.

Learning using the sampling methods in LP^{MLN}-learn was noticeably slower than pseudo-log-likelihood, as expected. It took 272 minutes over 50 iterations with 50 samples each iteration, or four-and-a-half hours.

Learning with any of the discriminative methods in Alchemy on the translated program also shows the large increase in computing time over pseudo-(log-)likelihood: fifty iterations of the MC-SAT learning algorithm takes 6 hours and 28 minutes. This is noticeably longer than the method in LP^{MLN}-learn, which can be explained by the larger number of MC-SAT samples that are automatically determined in Alchemy: 344 samples.

Other gradient methods in Alchemy (Voted Perceptron, Scaled Conjugate Gradient) take equally long on this dataset. That they take equally long indicates that they do not reach the convergence criteria any quicker, and that all iterations are needed for computing the weights.

We also performed ten iterations of every learning method, to test these against the models learned using more iterations. Ten iterations normally take around 45 minutes to finish.

The rules of Program 4.1 introduce cycles into the program, which makes learning using ProbLog impossible: ProbLog does not accept the input program. Without these rules, ProbLog does not have any models to learn against (compare the contrastive formulations in Chapter 3); this is precisely the role of the (hard) guessing rules (which allow to contrast the evidence with all stable models). Hence using ProbLog for learning these weights is not an option.

Training on the direct interpretation as an MLN formula in Alchemy was much faster than the LP$^{\text{MLN}}$-learn method again, taking 62 iterations and stopping because of convergence, in 26 minutes. When using Voted Perceptron for learning, 100 iterations were done without converging, over 42 minutes.

From the result in Section 4.3.2, we implemented a learning method in LP$^{\text{MLN}}$-learn which learns separately on each dataset, on each iteration, using MC-ASP. Since Alchemy performs the sampling and learning on every dataset separately each iteration, we implemented and tested this in the LP$^{\text{MLN}}$-learn program. Note that, as explained in Section 3.3.1, both ways lead to the same probability distribution in theory, with the difference that the number of samples is higher if performed on every scene separately. The new learning method takes 100 minutes, versus the three to four hours of the other method.

Outdoor For the outdoor dataset different problems arose with learning, under both systems. For the combined training and validation data, the LP$^{\text{MLN}}$-learn methods could not learn on this dataset: the MC-ASP method had not finished a single iteration overnight, and the Pseudo-log-likelihood did not finish a single iteration on even a smaller dataset of only ten scenes. This seemed not to be purely an implementation-specific problem, as Alchemy also failed to initialize the learning procedure: it could not load all ground Markov Networks into memory. Alchemy actually used fifteen gigabytes of swap space in addition to fifteen gigabytes of RAM. Pseudo-log-likelihood learning in Alchemy posed no problems, and took 30 minutes to perform on the full training data. The same problem arose in Tuffy, despite using PostgreSQL to save the groundings: it required more memory than available to compute all groundings. Alchemy required 22 hours for 39 iterations with only 200 MC-SAT samples each iteration per dataset (for 25 datasets)—the maximal we were able to compute due to memory problems—which is a lot less samples than the initial settings.

For this reason we use a much smaller number of scenes in the following experiments (10 or less), which we have found to not have a noticeable effect on the effectiveness of the methods. In addition, this keeps computation time down for our experiments to around an hour for 100 samples over 30 iterations.

Learning in LP$^{\text{MLN}}$-learn under our separate database implementation, on all scenes, posed no problems as encountered when learning on the combined data. This is the same result as encountered with learning on the indoor dataset, to a greater extent; learning was only possible when the stable models were not combined but learning was done separately.

4.3.2 General accuracy observations

The average accuracy over the validation dataset for the log-odds model is 0.54, versus 0.48 without using constraints and 0.49 using uniformly weighted constraints. This learning method is therefore a noticeable improvement over the local classifier and over all-equal weights. We will use this as a reference learning result for the other methods and learned models.

Pseudo-log-likelihood

As discussed in Section 3.2 and found in (Lowd and Domingos 2007) we expected Pseudo-log-likelihood to perform less well than the sampling (discriminative) methods.

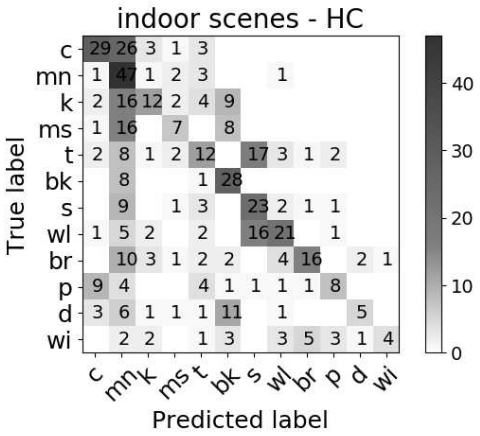
For PLL in LP$^{\text{MLN}}$-learn, getting the step size right proved important for prediction accuracy. At an initial learning rate of 10 (which is very big, and just used for testing purposes), the model performed poorly at 0.2 accuracy; at a learning rate of 1 or 0.1, performance is about 0.5 over 20 iterations. This decreased slightly over 30 iterations (to 0.49) which might be a case of overfitting. The predictions are also quite different for the different step sizes, see Fig. 4.1a) and Fig. 4.1b).

We tried using a decay in the learning rate[2], which after 10 iterations performed at only 0.32 average accuracy, which later improved to 0.43. Here the learning method first goes too far and overshoots the step (and classifies everything to a few classes), which it later tries to compensate; this is visible from the learned weights. However, because of the decay, it seems to fail to compensate properly.

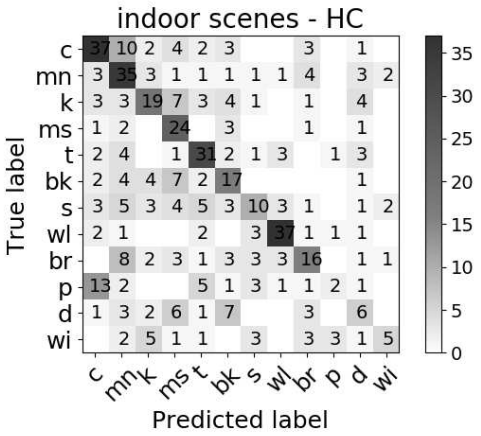
The sensitivity to the step size is the primary advantage and reason for using smarter methods which try to approximate the appropriate step size (such as line search). This also indicates that the step size is important to get right for LP$^{\text{MLN}}$-learn.

The Alchemy PLL-learning method is based on the L-BFGS algorithm, which approximates the Hessian to determine the search direction and uses line search for the step

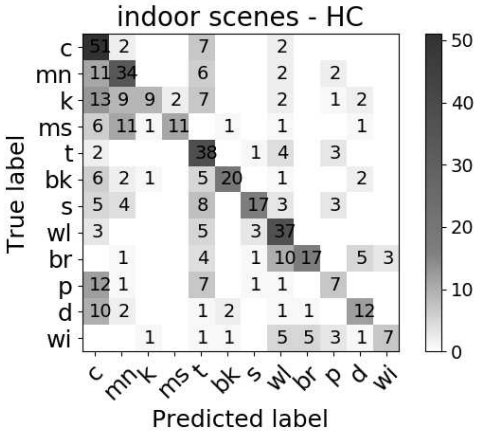
[2]The idea of the learning rate decay is that the algorithm first makes big steps towards the right settings, and tweaks this with smaller steps. Of course, this depends on not overshooting it in the first step; it also introduces a new parameter, the decay factor, which complicates learning.



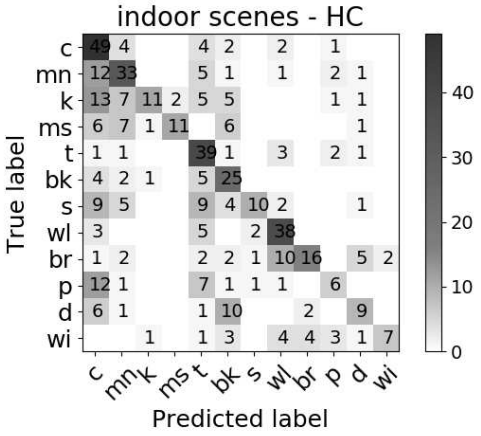
(a)) PLL LP$^{\text{MLN}}$-learn, step size 1



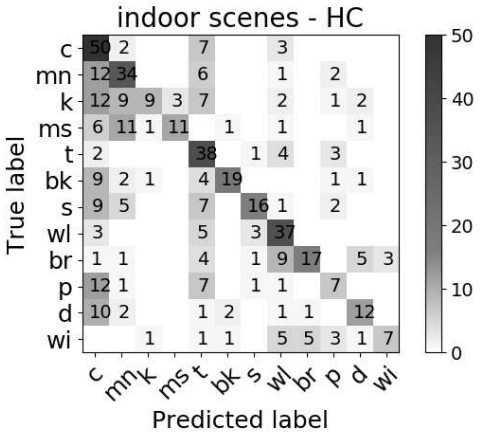
(b)) PLL LP$^{\text{MLN}}$-learn, step size 0.1



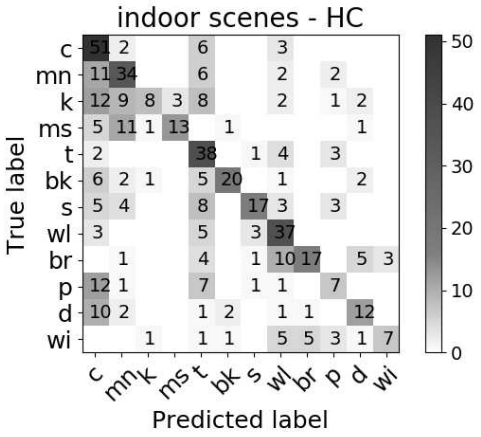
(c)) PLL Alchemy



(d)) MC-ASP LP$^{\text{MLN}}$-learn



(e)) MC-SAT Alchemy



(f)) MAP Alchemy

Figure 4.1: First models Indoor, validation results

size. Using the weights as output by Alchemy after appropriate scaling, performance was bad at 0.19. However, here the weights are centered around 0, with differing signs; after adding 1 to every weight so as to make all weights positive, this model performed at 0.53 on the validation data. This can be explained by the scaling required for inference in Clingo or DLVHEX: scaling makes the difference in weights bigger because of the differing signs, so shifting the weights before scaling prevents this. Alternatively, flipping the sign of the weights (which is here contrary to the semantics) also improved the prediction accuracy in the same way.

Some of the same classification patterns can be seen in the LP$^{\text{MLN}}$-learn and Alchemy learned pll models—too much classification to the "chair" and "monitor" classes—but the Alchemy model performs better overall.

This is an important limitation of weight learning in LP$^{\text{MLN}}$ as it currently is, where the precision of learned weights is difficult to take into account for inference. Here it is also important to note that inference in every step of the LPMLN-program is currently done by rounding of the weights, which drops important precision of the weights. In general, we have found that the scaling can have a very big impact on the predictions of learned models; issues arose especially when all weights were below zero (as was the case by models learned using Alchemy), where flipping the sign was necessary for good results. The predictions of the Alchemy model can be seen in Fig. 4.1c).

Besides the different gradient ascent algorithm used in Alchemy, there is another difference between the models which could explain the difference in performance between the Alchemy and LPMLN-Learn models. As described above in Section 4.1, the input LP$^{\text{MLN}}$ program has a conditional or discriminative strucure, in that the facts of the scene are input and only the weak constraints are learned. When performing the learning as such a (discriminative) PLL model—learning only the weights of rules involving "*assigned_label*" and "*not_assigned_label*" predicates—the model performed the same, and the corresponding prediction (on the validation set) is almost exactly the same as the PLL Alchemy model learned without specifying these predicates, at 0.53 average accuracy. Therefore, the difference in performance of the two algorithms is best explained by the different implementations (L-BFGS in Alchemy performing better) and not because of the way we formulated the input data for LP$^{\text{MLN}}$-learn.

MC-SAT/MC-ASP

For MC-ASP in LP$^{\text{MLN}}$-learn, we again first performed multiple tests on a small training-set of 4 scenes, to determine the optimal settings. We anticipate that the same results as in pseudo-likelihood based learning in LP$^{\text{MLN}}$-learn will occur, where differences in step size were noticeable. Furthermore, for MC-ASP the number of samples has to be established. Theoretically, if more samples are taken, the distribution sampled from is closer to the posterior (c.f. Section 3.2.3). Here the payoff is the learning time; more samples might not be necessary, if a few samples already gives enough information for the right direction.

Table 4.2: LP^{MLN}-learn MCASP, parameter tests

Val results MCASP settings			
Settings			
Iter; samples; learn rate	**0.1; 20; 20**	**1; 20; 50**	**1; 50; 20**
Avg. Acc.	0.48	0.42	0.54
Settings			
Iter; samples; learn rate	**5;20;20**	**0.1; 50; 20**	
Avg. Acc.	0.41	0.51	

From table Table 4.2 we can see that taking 50 samples per iteration performs better than performing more iterations, with the highest average accuracy (0.54) with a step size of 1 at 50 samples per iteration.

Training a model on the complete training set with these settings performed badly after 20 iterations, at 0.38 average accuracy. This did not improve after 30 iterations (0.35 avg. accuracy). Training with more samples at a learning rate of 0.01, over 50 iterations, performed poorly at 0.33 average accuracy; however, taking the average over all iterations performs at 0.41 accuracy. Using training rate of 0.1 and 100 samples performed at 0.45 accuracy. Taking the average weights over all iterations obtains 0.48 accuracy: equal to the local classifier (here scaling the weights too much made the results worse).

This bad performance of the MC-ASP method is not because of bad gradient ascent settings, but because of sampling problems on the combined data. This we found out after considering the difference with the following Alchemy learned models.

An initial model learned using Alchemy trained on the full training set with default settings, performs at 0.52 average accuracy. When proceedings as with LP^{MLN}-learn to test different parameter settings on a small dataset of four scenes, there were actually only little differences between models learned with learning rates of 0.01, 0.1, and 1, trained using 50 samples. All converged and halted learning due to no noticeable difference in weights after around 50 iterations; the only difference is how they reached this final joint weight setting. The learning rate of 1 first overshot and went back, whereas the 0.01 model slowly went to the best obtained weight setting. All had an average accuracy of 0.5 on the complete validation set, which was reached in all three models after around 10 iterations. Exactly the same results were obtained after using 20 samples; the final weights learned sometimes differ, but the end prediction results are equivalent. However, some models performed better after 10 or 20 iterations than after convergence; this can be explained by overfitting on the small training set.

When trying two different settings on the complete dataset, 0.1 and 1 for learning rates, we did obtain noticeably different results. The 0.1 learned model attained 0.51 on the validation set, whereas the model with learning rate of 1 attained 0.54. From these tests on Alchemy we can conclude that the difference in parameter settings is more important for bigger datasets, possibly because the learning method is not run to convergence. In

addition, it might converge faster with better parameter settings (or smarter methods such as those in Section 4.3.2). The results of the Alchemy MC-SAT model on a learning rate of 1 can be seen in Fig. 4.1e).

For completeness, although we will not discuss MAP-training further here in this chapter, we have also included a model based on learning using MAP in Fig. 4.1f). This model performed well, without tweaking the parameters, at 0.53 average accuracy.

From these initial tests we see that the LP$^{\text{MLN}}$-learn and Alchemy methods are not equivalent, or at least do not perform the same way on very similar learning methods (the sampling methods are almost the same). As mentioned earlier in this chapter, this led us to implement a weight learning method in LP$^{\text{MLN}}$-learn that samples per dataset, rather than on the combined data.

Training on our implementation that uses separate datasets for learning at every iteration, we used 20 samples per dataset over maximum 50 iterations, with (as before) a learning rate of 0.1. After about 20 iterations, the weights did not increase very much anymore, and the predictions and weights are similar to the results obtained on the full dataset using the combined data in LP$^{\text{MLN}}$-learn. However, performance is better at 0.48 versus 0.44. The slightly higher average accuracy could be due to the higher sample size, as 20 samples per dataset are taken for every one of the training scenes. We did not notice that taking bigger samples increased the accuracy noticeably in the standard method, however; proper testing of this was impossible because of the memory issues in learning. An important result also is that they are *not* equivalent in practice for learning, as learning on the separate stable models is noticeably faster than learning on the combined stable model by renaming of constants. This can be explained by the number of groundings needed in the combined dataset, which is bigger than grounding per dataset.

When learning with the same method using a step size of 1 in the gradient descent, we obtained slightly better results after 30 iterations, at 0.5 accuracy. At this stage, the prediction and learned model is again very similar to the model on the combined data. However, after 50 iterations, the result was worse at 0.42 accuracy. We could clearly see the weights oscillate strongly over the iterations. This led us to conclude that learning with a learning rate of 0.1 was more stable; the best result was still obtained training only on the smaller training set, rather the including all the training scenes.

The best result in LP$^{\text{MLN}}$-learn using MC-ASP sampling was obtained after training using our implementation, with a learning rate of 1 over 30 iterations, taking the average of weights over all iterations. This average prevents possible overfitting, or issues due to too big a step size. This performed at 0.52 average accuracy on the validation set. The results are visualized in Fig. 4.1d).

Different gradient ascent algorithms

As discussed in Section 3.1.3, the choice of algorithm for optimizing the gradient ascent can have a big effect on the effectiveness of a learning method. The improvements of

gradient ascent are meant to tackle precisely the problems encountered in the previous section with the step size of gradient ascent. We hypothesize that using these methods, with the same number of samples and iterations, should perform at least as well as the previous best MC-SAT method.

The methods in Alchemy that solve issues with standard gradient descent based on MC-ASP/MC-SAT sampling, are Diagonal Newton (DN), Scaled Conjugate Gradient (SCG) (possibly with a preconditioner, P-SCG), and using per-weight learning rates. DN and (P-)SCG tackle the problem of determining the step size and oscillating behaviour of standard gradient ascent, while per-weight learning rates are effective against ill-conditioned problems. The DN and P-SCG implementations in Alchemy uses line search to determine the step size (Lowd and Domingos 2007, p. 6), and the possible preconditioning in SCG transforms the optimization problem to also solve against ill-conditioning.

Our dataset did not seem very ill-conditioned, as the initial counts (with initial weights) are very close to one another and actually almost exactly equal for the initial weight setting. Taking the ratio of biggest versus smallest counts in the data in the first iteration is a way to approximate the condition number, as also performed in (Lowd and Domingos 2007). (Note that after many iterations this condition number can change due to different weight settings, so this is only an approximation.) For this reason we did not expect per-weight learning to lead to better results; we tested this by also learning a model using per-weight step sizes. This is only implemented in Alchemy for voted perceptron and contrastive divergence. CD, as discussed in Section 3.2, is identical to normal MC-SAT learning except for small sample sizes.

Using a preconditioner in P-SCG also has the purpose of reducing the condition number. It does this by performing a linear operation on the data before learning; in Alchemy, and as used by (Lowd and Domingos 2007) for MLN, this calculates the inverse diagonal Hessian to approximate the inverse Hessian, which is then multiplied with the data to reduce the condition number. Because of this, we also do not expect using a preconditioner to improve the prediction by a lot.

SCG performed badly out of the box, at an average accuracy of only 0.26. However, when flipping the sign of the weights the average accuracy went to 0.53. When using a preconditioner it also performed at an average accuracy 0.53, here the weights were in the right sign. Flipping the signs of the CD method also increased the accuracy to 0.53.

The need to change the signs of all the learned weights in this model is counterintuitive. Alchemy learns the weights of the negated input, and outputs the weights corresponding to the negated input. These weights are the negative of the weights of the input program, and therefore correspond to the weights which should be input for ASP with weak constraints. However, it seems Clingo only performs well if (most or) all input weights are positive and correspond to penalties; according to the semantics this is not equivalent (cf. Section 2.3.3). We also noticed that the factor used can significantly alter the predictions. Scaling with a positive factor is natural from the LP^{MLN}-semantics (Lee, Talsania, and

Y. Wang 2017a, p. 5) and using lower scaling factor decreases the difference in weights slightly. We have not been able to find out the reason why this sensitivity to signs, and the need to flip the signs, was necessary for some of the learned program: it is not behavior we encountered in all our tests.

4.3.3 Indoor Test Results

On the separate test set of 60 scenes, we tested the models of the following methods, with settings as determined in the previous section:

- Log-odds;
- Pll on initial and training + validation data in Alchemy;
- MC-ASP of LP$^{\text{MLN}}$-learn (step size 1, 30 iterations, 20 samples per datasets)[3];
- MC-SAT(step size 1, 20 iterations, 40 samples) of Alchemy;
- PSCG (20 iterations, 40 samples);
- Diagonal Newton(40 samples, 20 iterations).

The goal of these tests is to determine whether the discriminative, sampling based models that we consider perform better than pseudo-likelihood based models (as was the case in Alchemy for many datasets (Singla and Domingos 2005; Lowd and Domingos 2007)) and better than the log-odds calculation. We also include the test result of assigning the same uniform weight to all rules, although this is of course not a learning method and serves purely as a reference point.

Not included in Table 4.4 is the PLL as learned in LPMLN-Learn: the bigger model performed rather poorly after training on the entire training set, at 0.41 average accuracy. Here the smaller model of the previous sections performed better, at 0.53 accuracy on the test set. We do not have a good explanation for the worse prediction of pseudo-log-likelihood learning LP$^{\text{MLN}}$-learn compared to Alchemy, but we conclude that the Alchemy implementation performs smarter gradient ascent.

Table 4.3: Test results Image Classification–indoor

Test results

Method	**Uniform weights**	**Log odds**	**PLL**	**PLL-Val set**
Avg. Acc.	0.48	**0.56**	0.54	**0.56**
Method	**MC-ASP**	**MC-SAT**	**P-DCG**	**DN**
Avg. Acc.	0.51	0.55	0.53	0.54

[3]Taking the average of the weights, to combat possible overfitting

From the obtained results, we cannot say that learning on a bigger dataset consistently improved the accuracy performance of a learned model. See also Section 4.4 and Section 4.5 where learning on smaller datasets outperforms the model learned here. Important is that the only model that performs as well as the log-odds calculation is the pseudo-log-likelihood based model learned in Alchemy on the combined training and validation set. The log-odds computation did not improve when including more data; the small training set seems to already include all information for learning. A central result for all models, though, is also that all weight learning methods outperform assigning the same weight to all constraints.

The successfulness of pseudo-log-likelihood and, even more, log-odds, indicates that the dependencies and influences between these results are presumably very small; if present, ignoring these in learning seems not to have had any bad effect on the prediction result.

4.3.4 Outdoor Test Results

The model with all-equal-weight assignments performs at 0.58 average accuracy on the test set. This is almost equivalent (only marginally worse) to not using the weights and only the local classifier.

The log-odds calculation performs at 0.68 average accuracy in the hybrid classification setup on the test set. After training on only six scenes, the performance is almost equal at 0.674. This indicates that training on smaller datasets, as described earlier in this section, should not harm the effectiveness of the other models.

Training pseudo-log-likelihood in Alchemy performed better, at 0.7 average accuracy on the test set. As we could only train on smaller datasets for the sampling methods, we trained these models on 10 datasets.

We have been unable to learn models that performed well when using LP^{MLN}-learn, trying many different parameter settings. The differences found in the indoor dataset in using a smaller dataset, or different step sizes, all did not give good predictions for MC-ASP; the best was up to 0.5 average accuracy, which is still far worse than using just the local classifier (performing at 0.58).

The same holds for learning in Alchemy on the translated program, where the average accuracy on the test set was only 0.53. However, as we found out through testing in Section 4.4, learning on a direct interpretation of the LP^{MLN}-program as a Markov Logic Network significantly improves the performance at 0.71 average accuracy.

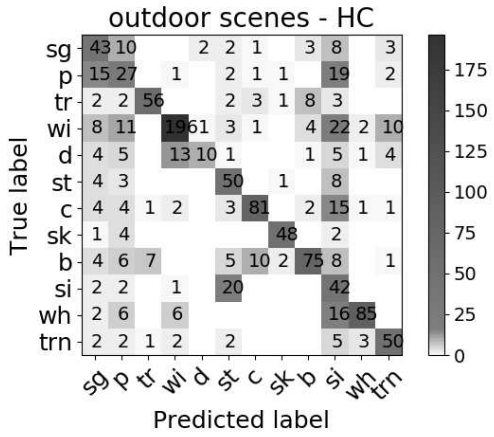
Confusion matrices of four models — log-odds, pll in Alchemy, MC-SAT on the MLN interpretation, and LPMLN — are in Fig. 4.2, and test results of the different models are in Table 4.4.

We conclude from these results that the sampling discriminative methods did not noticeably improve upon the pseudo-likelihood based method and the log-odds computation of

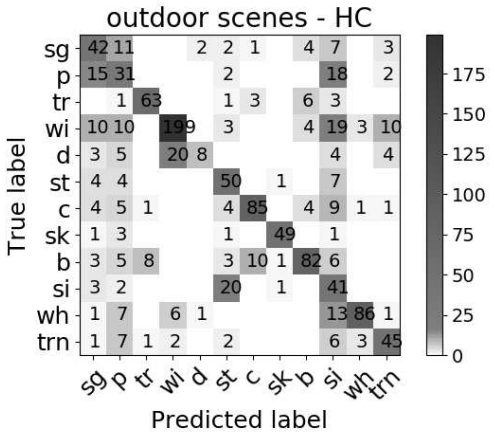
[4] Without scaling; scaling decreases the result.

Table 4.4: Test results Image Classification–outdoor

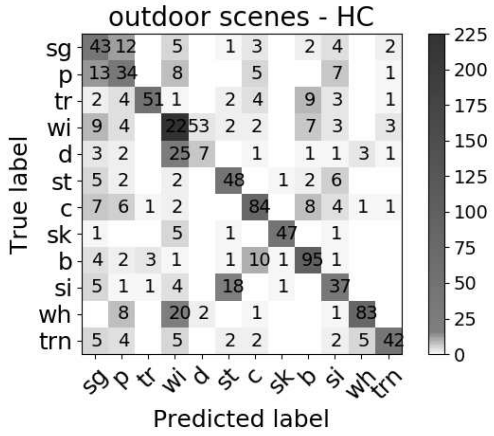
Test results			
Method	**Log odds**	**PLL-Alchemy**	**MC-ASP**[4]
Avg. Acc.	0.68	0.70	0.44
Method	**MC-SAT (Alch)**[4]	**P-DCG**	**MC-SAT(MLN)**
Avg. Acc.	0.53	0.5	**0.71**



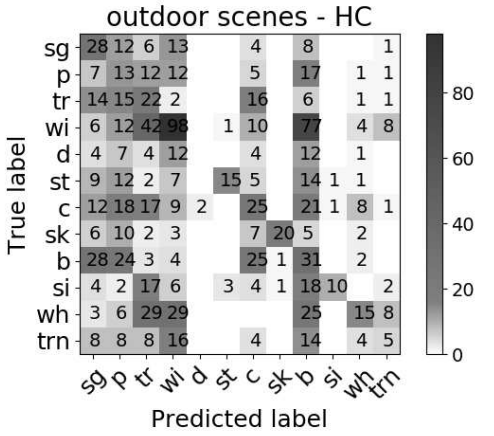
(a)) Log odds



(b)) PLL Alchemy



(c)) MCSAT - MLN interpretation



(d)) MCASP - LP^{MLN}-learn

Figure 4.2: Test predictions Outdoor dataset

the weights. In fact, on the outdoor dataset these latter models were the only ones that performed well, with the exception of pseudo-likelihood in LP$^{\text{MLN}}$-learn, which could not be used for learning. We therefore conclude that the effectiveness of the learning methods depends very much on the (structure of the) LP$^{\text{MLN}}$-program; in our case, the simpler objectives with stronger assumptions perform better. For reasons we have not been able to establish, the models learned using the discriminative sampling methods under the LP$^{\text{MLN}}$-semantics performed badly.

For models that performed well, changing the parameter settings did not significantly alter the predictions; for the models that didn't perform, we could not improve the results by changing parameter settings. What did significantly alter predictions was the scaling factor used for the weights, which is necessary only when learning in our test setup using ASP with weak constraints. If the scaling was too low, effects of the learned weights were not noticeable; if too big, it could to worse predictions. This is important both for weight learning and inference in LP$^{\text{MLN}}$, and it is immediately related to ASP solvers only allowing integer weights.

4.4 Independent constraints

As explained in Section 3.2.3, the log-odds computation can be seen as making a strong assumption of independence between constraints. Although this clearly doesn't hold in general—whether one constraint is satisfied or not normally has influence on other constraints—our results above indicate that for our dataset this property does, at least approximately, seem to hold.

For the outdoor program, this property seems *prima facie* to hold less (be a worse approximation) than for the indoor program. Consider the following constraint of the outdoor LP$^{\text{MLN}}$ program:

$$\leftarrow higher(x1, x2), assignedlabel(x1, C1), assignedlabel(x2, C2),\\ midsizedobject(C1), largeobject(C2), x1! = x2 \qquad : w.$$

This constraint contains two labels, whose values are intuitively also determined by values of the other constraints (similar to those in the indoor LP$^{\text{MLN}}$-program), and co-determines those truth-values based on the relative position of the two objects. We expect that another rule specifically there for assigning to the class *trunk*, for example, has effect on, and is effected by, the weight of this rule.

To elaborate, suppose an object $x2$ is assigned the label "trunk", due to its closeness to an object $x1$ labeled as "car". If the object $x2$ is big and above $x1$, the weight of the rule would be expected to be higher, due to its importance in prediction. However, if more are classified as windows by other rules (and supposing windows might or might not be large), and if this prediction is correct, this rule might be less important and so the weight less high. If this is true, the constraints of the outdoor LP$^{\text{MLN}}$ program are more

interdependent, with the additional expected outcome that this will make the log-odds computation less successful.

An efficient way of testing this would be to have a dataset where the rules are known to be correlated in a certain way, and test the effectiveness of the log-odds computation on such a dataset. Here "effectiveness" could both be taken as prediction accuracy as well as being able to capture dependencies. Here, we go another route: we retrain a model separately on all weights, by deleting all other soft constraints.

Note that learning this way does not just make the rules independent, such that a rule cannot influence whether or not another rule is satisfied. This could be achieved by renaming. It also has the effect of making the corresponding weights independent, meaning that per-rule learning rates are used. The log-odds computation also satisfies both independence properties. These tests, if the independence assumption is confirmed, can therefore explain the good performance of the log-odds training method.

4.4.1 Outdoor

As mentioned above, we first failed to train a good model in LP^{MLN}-learn. Contrary to expectations, learning on the separate rules, the average accuracy increases to 0.66 on the validation set. The model is trained on 10 datasets with a step size of 0.1, 20 samples per dataset, over 20 iterations. Here again the weights were in the wrong sign; we interpret these as good results nonetheless. On the test set, the average accuracy was 0.69. The same model learned on the combined rules, performed at 0.25 average accuracy with the same scaling factor.

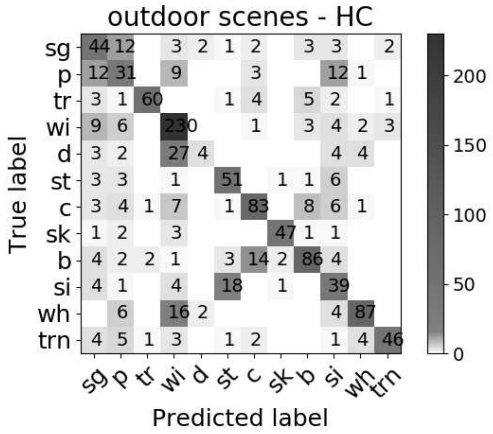
Training in Alchemy on the separate rules, on ten datasets, took 6 hours to train with 20 samples per rule per dataset, over 20 iterations. This means that there is no difference in learning time between learning all constraints together, and learning for *one* individual constraint. This performed at 0.43 on the test set. Here we found out (by accident), that a model trained on a direct interpretation of the LP^{MLN} program as a Markov Logic Network performed much better, at 0.69 average accuracy. The following results are of a direct interpretation as an MLN program. Here we used a learning rate of 1, with 20 MC-SAT samples per dataset over 20 iterations, which performed stably when trained on 10 datasets. Learning is also much faster than on the translated model: training the rules separately, training takes only 15 seconds for the entire MC-sat optimization for every rule, with the exception of the rules given in the previous example which involve two first-order evidence atoms, here learning takes 7 minutes.

The learned weights from both the dependent and independent models are in Table 4.5. From this table, one can see that the weights differ noticeably among the two models. However, the average accuracy on the validation data of the independent model is almost the same as that of the standard model, at 0.68 on the validation set.

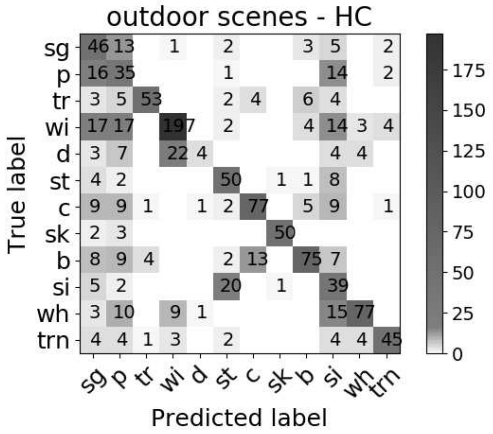
On the test set, the average accuracy over all scenes of the model with independent constraints is 0.72. With the model learning on the combined program, the average

Table 4.5: Dependency test weights– Outdoor dataset, MLN

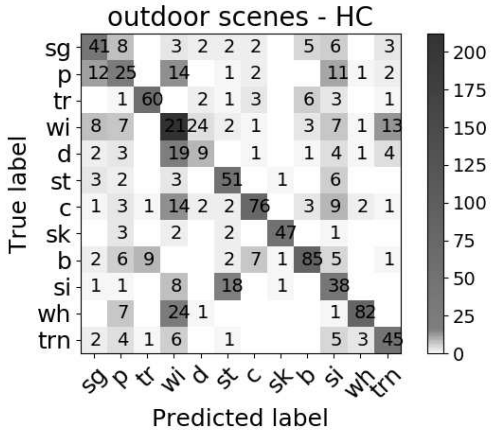
Dependent weights	**Independent weights**
9.74441	2.64852
8.73788	1.33428
9.74351	2.66882
9.74628	3.35119
9.74613	2.90921
9.74292	2.57161
8.99267	1.47502
9.74351	1.73687
9.74503	2.41325
9.74841	3.10189
9.74611	3.18317
9.74824	3.04824
9.74717	3.34722
9.74609	3.3889
9.74826	3.00206
0.246713	0.013945
0.241465	0.114842
0.206372	0.269905
9.74438	2.58067
5.84232	103.619



(a)) Independent constraints-MLN model

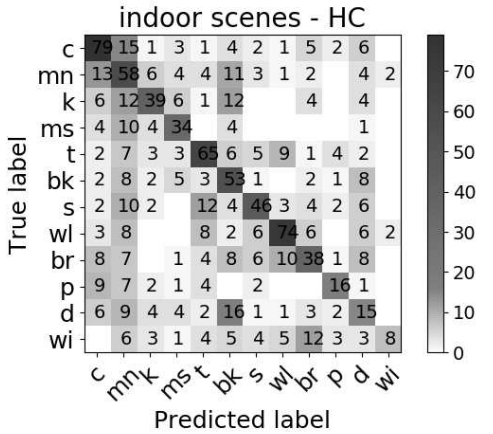


(b)) Dependent constraints-MLN Model

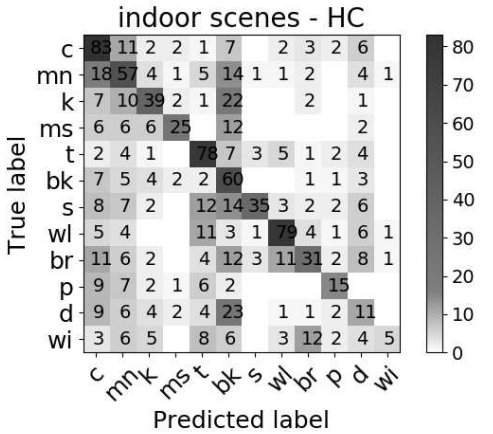


(c)) Independent constraint-LPMLN

Figure 4.3: Independent constraints learning, outdoor test results



(a)) Independent constraints indoor-LPMLN



(b)) Dependent constraints-indoor-LPMLN

Figure 4.4: Independent constraints learning, indoor test results, LP^{MLN}

accuracy on the test set is 0.71. The confusion matrices are shown in Fig. 4.3. From these we can see that, although the classifications of the independent constraints are more spread out, the classifications into wrong classes are less extreme than when learning all rules simultaneously, and thus allowing dependencies to be taken into account.

4.4.2 Indoor

Building on this result, we then also tried learning an independent model of the indoor dataset, using the same settings. Here we used the translated program in Alchemy for training, with settings as in Section 4.3.3, except using 30 samples each iteration per dataset. To keep learning time manageable, we learned on 10 scenes per program.

After getting the right scaling factor on the validation set, we obtained 0.54 average accuracy on the test set. This is only slightly worse than the full model. Because of one negative weight for a rule (all others were positive), which could be because of the bigger step size, we retrained another one with a step size of 0.1, using twenty scenes. We did this to combat possible overshooting (overcompensation by step size in gradient ascent) of weights and to see if the size made the standard method perform better. This model performed almost equivalently, at a test average accuracy of 0.53, which means no improvement.

The LP^{MLN}-learn model, with a learning rate of 0.1 and 20 samples over 20 iterations, per rule, performs at 0.52 average accuracy on the test set. This is a minor improvement over the combined model. Looking at the predictions in the confusion matrices Fig. 4.4, we can see the predictions are very similar (despite the difference in calculated weights).

We conclude that in both our datasets learning on constraints independently performed as well or better than learning on the combined programs. In the indoor dataset, the difference in prediction is only slight and the predictions are very similar. The more noticeable result was found in the outdoor dataset, which we first expected to have more dependencies among the constraints. Among the Alchemy learned models, although the predictions were very similar, accuracy was higher in the independent model: the model learned on independent constraints outperformed the standard model by a big margin. This indicates that our input programs largely have an independent structure, even though this was anticipated to not hold for the outdoor dataset. Moreover, this could help explain the good performance of the log-odds learning method, as observed in our previous tests.

4.5 Missing data

As mentioned in Section 3.3.3, there are multiple scenarios under which data can be missing. In this section, we will create different datasets corresponding to the different scenarios discussed, by adapting the data used throughout this chapter in a controlled manner. That is, we consider MCAR, MAR, data that is not-MAR, and a dataset which

we constructed for the purpose of a sanity check, to consider the performance of learning methods in the case of missing data.

Both Alchemy and LP$^{\text{MLN}}$-learn allow for data imputation using EM; our hypothesis is that these implemented algorithms will work better than the standard gradient ascent implementations in the case of randomly missing data. To be more precise, if data is missing completely at random, we hypothesize that this will severely alter the accuracy of normal learning methods; if the observation model is more specifically setup so objects belong to specific classes are missing, we hypothesize that imputing the missing data will improve the accuracy.

Aside from testing on the accuracy, we also consider the per-class predictions by looking at the confusion matrices to investigate whether predictions to specific classes are hindered by deleting data. This will give more specific information about the predictions than can be obtained from only the average accuracy over all classes (and over all scenes).

To test our hypothesis, we constructed multiple datasets from our initial data for both indoor and outdoor. In the first we deleted 20% of the data per scene completely at random (MCAR). In the second, for the indoor dataset we deleted 50% of objects that are observed above walls randomly, and another 25% of objects that are contained in walls randomly. (If an object is both above and contained in a wall, we only select it if is not deleted by any of these rules.) Note that this deletes also any atoms containing any objects, so there is a possibility that a lot of information is lost, depending on the specific scene. This is meant to simulate camera shots where the camera is slightly pointed downwards, so that higher objects are not observed. As one can see in Program 4.5, objects contained in walls and above walls are present in a number of rules: we expect learning these rules to be hindered by the missing data.

Program 4.5

$$
\begin{aligned}
&\leftarrow assignedlabel(x,c), contained\_in\_wall(x) && : w.\\
&\leftarrow assignedlabel(x,t), contained\_in\_wall(x) && : w.\\
&contained\_in\_wall(x2) \leftarrow contains(x1,x2), assignedlabel(x1,wl) && : \alpha.\\
&\leftarrow assignedlabel(x,br), in\_upper\_part(x) && : w.\\
&\leftarrow assignedlabel(x,wi), in\_upper\_part(x) && : w.\\
&\leftarrow assignedlabel(x,s), in\_upper\_part(x) && : w.\\
&in\_upper\_part(x2) \leftarrow under(x2,x1), assignedlabel(x1,wl) && : \alpha.
\end{aligned}
$$

For the outdoor dataset, we deleted objects according to a similar method, deleting objects that are below a car or below a building, with a 60% chance for both.

Note that by construction the value of the missing atoms is independent of the observation value of these atoms, given the value of the observed atoms (namely, walls). Hence the

MAR criteria of Section 3.3.3 still holds. We use 10 datasets for both our tests, both indoor and outdoor.

Furthermore, we constructed two missing-data datasets (for the indoor and outdoor dataset) by deleting objects with a 90% chance. This is data not missing at random (NMAR), as the class of the deleted object determines whether it is observed or not; this can be interpreted as deleting data because you do not like the outcome. Here, knowing the observability mechanism will give information on the value of the hidden variables. For the indoor dataset we remove monitors, keyboards, mice and tables — all related to the "ms" (mouse) label in the LP$^{\text{MLN}}$ program. From the outdoor dataset we deleted cars and trees, as these are involved in many constraints.

As a sanity check, we deleted only certain atoms in the dataset: atoms saying that objects are assigned the label "mouse" or "keyboard" for the indoor dataset, and "window" or "building" for the outdoor dataset. As this keeps the facts intact but removes the evidence required for learning these classes, this should have the effect of making predictions for the mentioned classes noticeably worse.

Since we did not change the parameter settings — these tests have been performed in Section 4.3.2 — we tested the learned models immediately on our test set.

4.5.1 Data MCAR

The average accuracy on the indoor for the missing completely at random data, was 0.55 for the non-EM method and 0.45 for the EM-model in Alchemy. In LP$^{\text{MLN}}$-learn, it was 0.55 for the standard learning algorithm, with settings as in Section 4.3.3. Using the EM algorithm, the average accuracy was the same at 0.55. Between these two LP$^{\text{MLN}}$-learn models, we can see only very little difference in the predictions.

For the outdoor dataset, In LP$^{\text{MLN}}$-learn, the standard MC-ASP algorithm performed at 0.47, versus 0.45 for the Expecation Maximization algorithm. This is in line with the results in Section 4.3.4; LP$^{\text{MLN}}$-learn fails to learn good models on the outdoor dataset. The same holds for the Alchemy sampling-based model, which underperforms at 0.54. The resulting predictions are very similar to those in Section 4.3.4. This is also the case for a learned model using pseudo-log-likelihood in Alchemy, performing at 0.7. From this we conclude that in the case of data missing completely at random, normal gradient ascent methods work as before.

4.5.2 Data MAR

Because of the selective nature of the observation mechanism in the constructed MAR data, we anticipated that standard methods would fail here and EM would be required for successful learning.

Indoor The average accuracy on the MAR-constructed indoor dataset was 0.49 in LP$^{\text{MLN}}$-learn, almost the same as the model learned using the complete scenes. This is

```
:∼ assigned_label(x,c), contained_in_wall(x). [278,x]
contained_in_wall(x2) :- contains(x1,x2), assigned_label(x1,wl).
:∼ assigned_label(x,t), contained_in_wall(x). [270,x]
contained_in_wall(x2) :- contains(x1,x2), assigned_label(x1,wl).
:∼ assigned_label(x,br), not contained_in_wall(x). [257,x]
contained_in_wall(x2) :- contains(x1,x2), assigned_label(x1,wl).
:∼ assigned_label(x,wi), not contained_in_wall(x). [405,x]
contained_in_wall(x2) :- contains(x1,x2), assigned_label(x1,wl).
:∼ assigned_label(x,c), not in_lower_part(x). [445,x]
in_lower_part(x1) :- under(x2,x1), assigned_label(x2,wl).
:∼ assigned_label(x,br), not in_upper_part(x). [183,x]
in_upper_part(x2) :- under(x2,x1), assigned_label(x1,wl).
:∼ assigned_label(x,wi), not in_upper_part(x). [187,x]
in_upper_part(x2) :- under(x2,x1), assigned_label(x1,wl).
:∼ assigned_label(x,s), not in_upper_part(x). [-113,x]
in_upper_part(x2) :- under(x2,x1), assigned_label(x1,wl).
```

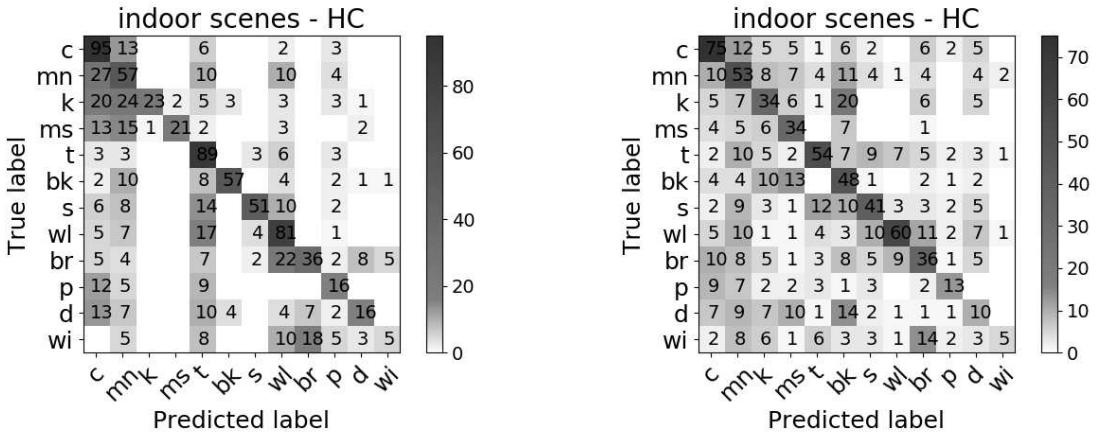
Figure 4.5: Learned weights, MAR data indoor

surprising, as the size of the data after deleting atoms is sometimes less than a half of the original size. In Alchemy, the EM method performs at 0.55 average accuracy over the test set, which is the same accuracy as the non-EM (standard) method. The Expectation Maximization method performed much worse at 0.37 accuracy.

Because of this surprising stability and performance of non-EM methods, we removed more objects and atoms with a 90% resp. 50% chance, the method as explained above. In LP$^{\text{MLN}}$-learn, the non-EM method performed at 0.47 average accuracy, where the EM learning method performed almost equivalently at 0.46. In Alchemy, the average accuracy of the standard algorithm was again high at 0.54. This means that the average accuracy does not decrease significantly when a lot of data is missing from the scene. Looking at the predictions as represented in the confusion matrices Fig. 4.6, we can also see that there is no label that was failed to be learned properly, even if there is hardly any evidence for a specific rule. In particular, according to our hypothesis, we would expect classes present in Program 4.5 to be less well predicted. However, both matrices in Fig. 4.6 show good results for these classes, and are largely the same as Fig. 4.1e).

This is also visible in Fig. 4.5 which is a part of the learned LP$^{\text{MLN}}$ program. As one can see, the rules with deleted objects also have learned weights. In Fig. 4.5 the signs are flipped to fit the semantics of ASP with weak constraints). The last rule does have a negative resulting weight, but this does not severely affect the resulting predictions. This negative weight also occurred in previous models for the indoor dataset in LP$^{\text{MLN}}$-learn.

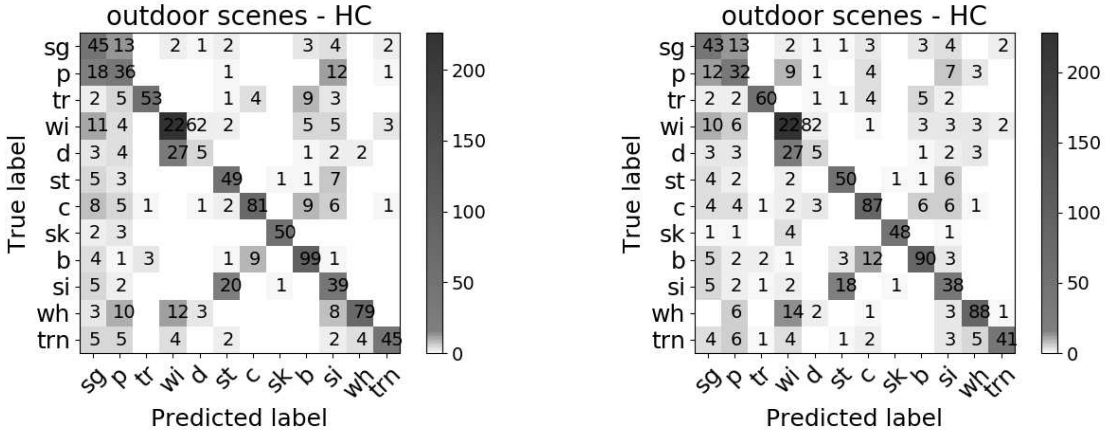
As a further test, we performed the same method for deleting with a 70% and 40% chance of being deleted, for objects contained in or under both walls or tables, respectively. This



(a)) Mar indoor–Alchemy

(b)) Mar indoor–LP$^{\text{MLN}}$-learn

Figure 4.6: MAR missing tests, indoor results



(a)) Mar outdoor - 80% test

(b)) Mar outdoor - full dataset

Figure 4.7: MAR missing tests, outdoor results

gave almost identical predictions to those in Fig. 4.6, indicating that this result is not because of the specific classes we took into account when deleting the data.

One explanation for the effectiveness in the case of missing data, is that the LP$^{\text{MLN}}$ program has multiple constraints that have the same effect on assigning labels. For example, the following two constraints

$$\leftarrow assigned\_label(x,s), not\, contains\_book(x) \quad : w.$$
$$\leftarrow assigned\_label(x,s), not\, in\_upper\_part(x) \quad : w.$$

both regulate the probability of assigning objects as shelves. It seems then, that deleting objects that are high does not influence the effectiveness of learning in the indoor scenes; the other constraints manage the predictions, making up completely for the constraints which are affected by the missing data. If this explanation is correct, deleting rules based on their class (not-MAR data) would have a noticeable effect.

Outdoor On the outdoor dataset, as in Section 4.4 and Section 4.3.4, we did not manage to train well performing models in Alchemy (0.48 avg. acc.) on the translated program and in LP$^{\text{MLN}}$-learn (0.45 avg. acc.), as expected from those results. Because of these bad models, it is especially difficult to interpret the performance in the case of missing data (these did not improve). However, learning (as in Section 4.4) on the direct-interpretation MLN in Alchemy, we obtained 0.72 average accuracy on the MAR outdoor dataset. This was made worse by using EM at 0.64. This is therefore the same result as on the indoor dataset.

In the same way as for the indoor dataset, we then proceeded to delete up to 80% of objects below cars or buildings. This did again not have the expected effect; the average accuracy remained at 0.72. Again, the predictions of all classes are largely intact, as can be seen in Fig. 4.7. What we did notice though, is that learning time increased significantly on this missing-data dataset. We explain this by there being more assignments possible in the case of missing data, which is due to the rules in Program 4.3.

This high average accuracy over severely impaired scenes, is surprising. We would expect here that "streets" could not be properly predicted; looking more closely at the predictions per class shows otherwise, where streets are also very well predicted.

For both the outdoor and indoor dataset we found that deleting many objects based on a Missing At Random observation mechanism did not influence the learning effectiveness. Using the standard methods that performed well on the normal data, we obtained almost equivalent prediction results after deleting a significant number of evidence atoms. Furthermore, using EM actually made the learning methods perform significantly worse; these methods seem to not be fit for our specific dataset.

4.5.3 Data NMAR

On this dataset, since there is hardly any information for two classes, we anticipate learning to be severely affected. For the indoor dataset, we specifically chose the atoms deleted such that the "ms" (mouse) class could not be learned properly. However, the predictions using LP$^{\text{MLN}}$-learn are only slightly worse at 0.48 average accuracy, and the "ms" class was also well predicted. We obtained similar results when learning in Alchemy, at again 0.54 accuracy; the "ms" class was very well predicted.

For the outdoor, the same result was obtained, where the Alchemy model performed at **0.73** average accuracy. Note that this is the highest performance of all our tests; we expected that this model would perform noticeably worse. This result is then completely against our expectations.

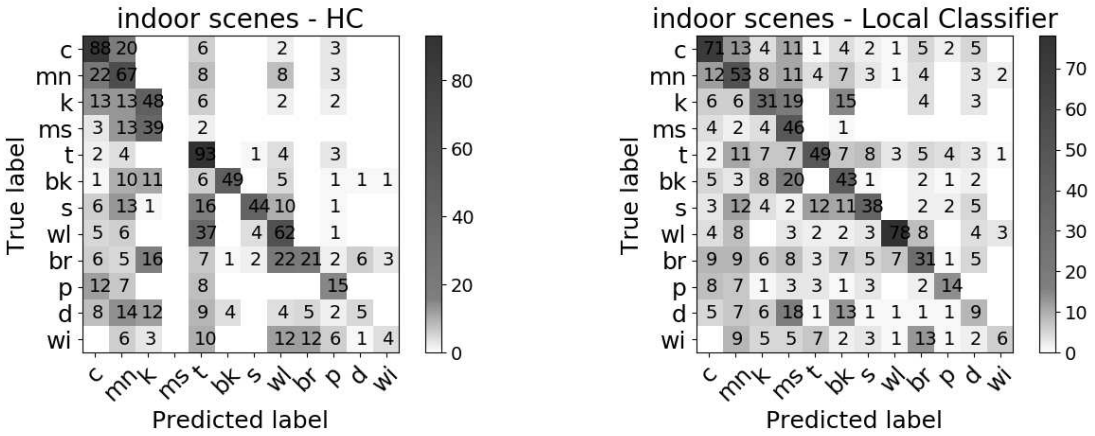
Our explanation of the good prediction of the MAR dataset—that multiple weak constraints manage predictions to the same classes—seems to be wrong, in light of these results. If this explanation held, deleting rules specifically based on classes should make all rules corresponding to that class perform badly in prediction, with the learning methods unable to properly learn the corresponding weights. However, we obtained the opposite result, and we therefore do not have a good explanation of the high effectiveness of our results on missing data.

4.5.4 Sanity Check

In light of the results obtained in this section, performing the sanity check is necessary for a better understanding; in the "sanity check" data certain label assignments are completely missing in the evidence. On the indoor dataset, training using the LP$^{\text{MLN}}$-learn system, the average accuracy on the test set is 0.49. This is only slightly worse than the results in Table 4.4. However, a more noticeable effect is visible in the per-class predictions, where there are no assignments to the "mouse" class. The "keyboard" class is still reasonably well predicted, although noticeably worse than the normal model. This is visible in Fig. 4.8a). Note that the bad prediction to the "mouse" class is not because of few predictions to this class from the local classifier: see Fig. 4.8b).

For the outdoor dataset we again trained using the direct MLN interpretation in Alchemy, which performed at an average accuracy of 0.73 (0.727). This is again very high, and we could not notice any clear effect on the "window" and "building" classes, where the predictions of the LP$^{\text{MLN}}$ program improved upon those of the local classifier. To further test this, we proceeded to remove entirely the two rules involved in the prediction of the "window" class. Here the accuracy dropped to about equal to the local classifier with 0.59 average accuracy, and a strong overclassification to the "window" class. These predictions for this class are similar to that of the local classifier alone. Hence learning without evidence seems to have improved the performance drastically.

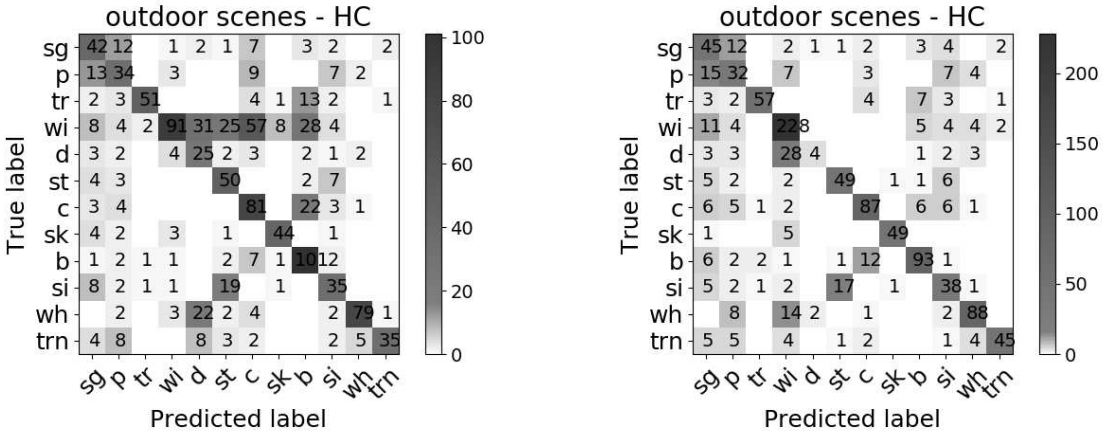
This phenomenon was also present on the indoor data when removing the constraints related to classifying objects belonging to the "ms" class. Here a lot of objects were



(a)) Sanity check indoor

(b)) Indoor local classifier test result

Figure 4.8: Sanity check indoor results



(a)) Sanity check outdoor

(b)) Not-MAR outdoor

Figure 4.9: Sanity check and NMAR outdoor result

classified as mice; noticeably more than the local classifier alone, too. This means learning with no related evidence atoms does not have the same effect as not including relevant rules.

We proceeded to delete the objects corresponding to the least represented classes (in the test set): "door" and "person" instead of the classes of the previous paragraph. Here the prediction accuracy dropped to 0.59; the "door" class especially is not well predicted. However, it is important that it is not underrepresented but *overrepresented* in the prediction. This is not noticeable for the "person" class. See Fig. 4.9 for the result.

We therefore have conflicting results for the indoor and outdoor dataset in these tests. This result cannot be explained by few predictions to these classes from the local classifier, as shown in the result on the indoor dataset. In the tests on the outdoor dataset, we explain the over-classifications with all the other weights being penalties, whereas the "window" class, say, does not have a penalty so an object is easily classified as such. However, for the indoor test the "mouse" class was underrepresented, meaning that the learned weight (as penalty) was too high.

Summary Against expectations, deleting a lot of data did not severely impair learning: the predictions of the learned models in multiple missing-data scenarios were still quite accurate, up to very accurate. We have been unable to find a good explanation of the consistent performance, expecting that the performance would be bad on the N-MAR and sanity check data, if nothing else. In the "sanity check" cases where we deliberately tried to force a malfunctioning model, we did notice worse performance, but even here this was unexpected and learning could still perform better than not learning the weights.

We expect these results to be very particular to our hybrid classification setup and we do not expect these results to generalize. We only conclude that learning on data with missing, or entirely without, evidence gives unpredictable behavior that we have not been able to explain properly.

4.6 Overall results

Going back to the experiments and hypotheses listed in the beginning of this chapter, we can summarize our results as follows:

H1. MC-ASP/MC-SAT sampling methods do *not* outperform pseudo-likelihood based learning and the log-odds calculation on our LP^{MLN} programs.

H2. Different parameter settings influence prediction results of learning methods somewhat, but not by a huge margin. It is difficult to draw strong conclusions here, as the differences could only be tested on the indoor dataset.

H3. The constraints of our dataset are mostly independent; learning independently significantly improved the quality of the LP^{MLN}-learn model on the outdoor dataset.

H4. Using Expectation-Maximization (EM) does *not* outperform standard MC-ASP/MC-SAT learning methods in the case of missing data. Nor do standard methods fail to learn good models in the case of missing data.

Furthermore, we found that including more scenes does not consistently improve the quality of the learned models. For the indoor dataset using MC-ASP in LP^{MLN}-learn for learning, learning on fewer scenes improved the quality of the model. This better performance is surprising: in general, we conclude that for our input program and

datasets, a few scenes already contain all the information needed for learning. This is also reflected using log-odds, where calculating the weights from more scenes does nothing for the performance of the model. Lastly, learning on the direct MLN interpretation of the LP$^{\text{MLN}}$ program on the outdoor dataset significantly improved the quality of the learned model. This interpretation leaves out some dependencies and constitutes a simpler Markov Network, which makes learning easier and faster.

As regards the implementations, we encountered noticeable differences: LP$^{\text{MLN}}$-learn frequently produced poorer models than the once learned using Alchemy. Since the optimization algorithms of Alchemy are more sophisticated and tackle problems with standard gradient descent (e.g., l-BFGS, line search), we explain the better performance to this aspect of the implementation.

Other important results concern learning on independent constraints, and learning using a direct MLN interpretation. Learning on the independent constraints performed much better in the outdoor dataset for LP$^{\text{MLN}}$-learn, which we explain by the independence between the constraints of our LP$^{\text{MLN}}$-program. We also take this to explain the effectiveness of the log-odds calculation, which is based on this independence assumption.

The high effectiveness in the direct Alchemy interpretation for the outdoor dataset is also unexpected, seeing how bad learning in Alchemy on the translated program performed. The direct interpretation removes some implications and has a simpler structure, which made learning perform much better here. This is likely to be related to the result of the previous paragraph.

Perhaps the most surprising result is the high effectiveness of learning in the case of missing data. On the outdoor dataset, the best performing model was the one trained on a dataset with a lot of data deleted, data which violated the MAR condition. In this case, one would expect that the model could not be learned properly; let alone perform the best of all tested models. On datasets where we completely deleted all assignments to particular classes, the results did not clearly show a pattern. Here the weights still influenced the results noticeably, and not generally in a bad way. In fact, the best performing model on the outdoor dataset was learned on missing data, with an accuracy of 0.73.

4.7 Discussion

We do not anticipate that these results are general for weight learning in LP$^{\text{MLN}}$, rather that they are specific to our input program and setup. This goes especially for the results concerning learning on missing data, and results concerning the independence among the constraints or rules.

A conclusion that can be drawn is that the structure of the program is of central importance for choosing the learning algorithm for an LP$^{\text{MLN}}$-program. In our program, the structure is such that there few dependencies between the rules—as indicated by the results on independent learning—and only few weights to be learned. If an LP$^{\text{MLN}}$-program

has a different more complex structure, with more rules that are tightly dependent on one another, we anticipate more data will be needed and taking account of dependencies in learning is necessary for good prediction performance.

However, given that there is only one query predicate in both our programs, it is surprising that the discriminative methods did not manage to learn better here. For future research, it would be interesting to consider in further detail the impact of structure on learning methods. The results of programs trained on missing data are also surprising. A possible explanation for the performance here is the overparameterization in our programs (many rules doing the same thing), but more research is needed to confirm this.

We expect that for similar programs—programs with few weights and few dependencies among rules—the log-odds calculation is not a bad approximation for learning. Nor is learning in the direct MLN interpretation overall a bad starting point, as seen from the results in this chapter. However, a direct interpretation of an LP^{MLN}-program as a Markov logic formula, or even using a translation of a non-tight program as seen in Section 4.3.2, the MLN interpretation of an LP^{MLN} program can lose some vital dependencies. This again depends very much on the input program.

Lastly, based on the theory of learning on missing data, we do not expect the results of Section 4.5 to generalize to other datasets, and it is difficult to draw conclusions from these results. However, it does show that the performance on missing data is difficult to predict: again, our best performing model was learned on data Not Missing At Random (NMAR). A more detailed study on missing data would have to be done for a good account of the performance on missing data.

CHAPTER 5

Conclusion

In this work we investigated supervised weight learning in LP^{MLN} and compared different learning methods in different scenarios. Although weight learning in LP^{MLN} is a very difficult problem—in general, at least as hard as inference in LP^{MLN}, which is at least $FP^{\mathbf{NP}}$-hard—many efficient approximations for weight learning exist. These approximations are either general approximations of weight learning on Markov Networks, or specifically constructed for weight learning in LP^{MLN} such as the MC-ASP sampling method.

As we have seen in our experiments, even learning methods which make very strong assumptions on the distribution represented by an input LP^{MLN} program can work well in practice. This was so even for a simple log-odds calculation on our example input programs. From our experiments we conclude that no (class of) method(s) generally outperform(s) other method(s). Therefore, the best choice of learning methods depends on the particular structure and characteristics of the LP^{MLN}-program. Whether methods which make strong assumptions on the distribution will work properly depends on the LP^{MLN}-program whose weights are to be learned; we expect that some methods which performed well for us, will be poor learning methods on input LP^{MLN}-programs with a different structure. Furthermore, we found that missing data does not necessarily impede learning, and that the performance on datasets with missing data is hard to predict beforehand.

Some of these results can be expected from the theory of weight learning, where the structure and size of the dataset can explain the performance of the generative methods; however, some results were very unexpected. For this reason, to get a good grasp of the performance of learning methods in LP^{MLN}, more experiments on different input programs would have to be performed. We expect that some of our results, most notably those on independent constraints and missing data, do not carry over to other LP^{MLN} programs with different structure and different characteristics. To draw stronger conclusions on

general weight learning, more experiments on different LP^{MLN} programs will have to be performed.

As regards the specific systems available for learning, although we have not been able to use ProbLog, Alchemy performed best in our tests, both in speed and resulting accuracy. This is dependent on the possibility of a translation of the LP^{MLN}-program to a Markov Logic Network and so is very unlikely to hold in general. This was also seen in the example dataset used in Section 4.3.2. However, we ran into difficulty in using Alchemy 2 and related systems which we could not resolve. For native learning and for programs that do not allow a direct translation to a MLN, LP^{MLN}-learn is a good option for learning; improvements on the system, for example with smarter gradient ascent algorithms, are likely to improve the result of learning using the system.

As LP^{MLN} is a relatively new formalism, it has so far not been extensively used for collective classification problems or related problems, nor has weight learning for these applications been extensively studied. In this work we have not been able to test weight learning on different LP^{MLN} learning problems, also because creating meaningful LP^{MLN} programs for classification is a complicated and time-consuming process. It will be interesting to see how weight learning in LP^{MLN} behaves on different input programs. Furthermore, many methods described in Chapter 3 have not been tested, where one of the most interesting is probably lifted learning, which could make learning significantly faster. In addition, other methods not discussed in this work still exist. Investigating these methods is topic for further research. We hope the work done in this thesis is fruitful for further research, and the initial results a good starting point for further inquiry into weight learning in LP^{MLN}.

List of Main Symbols

Π	Logic Program
$\overline{\Pi}$	Unweighted LP^{MLN} Program
$\Pi_{\mathcal{I}}$	Rules R in Π s.t. $\mathcal{I} \models R$
$\Pi^{\mathcal{I}}$	Gelfond-Lifschitz Reduct of Π w.r.t $\mathcal{I}$
$\mathbb{L}$	Markov Logic Network
$\mathcal{I}$	Herbrand interpretation
$\mathcal{I}_{\boldsymbol{A}}$	Interpretation restricted to the values in $\boldsymbol{A}$
$\Im$	Set of Herbrand interpretations
$\mathcal{D}$	Dataset
w	Weight parameter
$\boldsymbol{w}$	Vector of weight parameters $(w_1, ..., w_k)$
P_Π	Probability function defined by Π
W_Π	Unnormalized weight function defined by Π
$Z, Z(\boldsymbol{w})$	Partition function (constant resp. function of $\boldsymbol{w}$)
$\mathcal{T}$	Transition function over interpretations
$\mathbf{X} = \{X_1, X_2, ..., X_n\}$	Random Variables
$\mathbf{X} = \{x_1, x_2, ..., x_n\}$	Outcomes of random variables
$\boldsymbol{E}_{\mathcal{D}}$	Empirical expectation over $\mathcal{D}$
$\boldsymbol{E}_{\boldsymbol{w}}$	Expected sufficient statistics
σ	First-order signature
x, y, z	Variables
p, q	First-order predicate symbols
$a, b, c, ...$	Constant symbols
$f, f_1, ...$	Function symbols
$A, A_1, ..., B, B_1, ...$	First-order atoms
$F, F_1, ...$	First-order formula
R	ASP rule
w	Weight of ASP rule
$\overline{A}$	Classical Negation $\neg$ of atom A

Bibliography

Alchemy: Open Source AI (2019). URL: http://alchemy.cs.washington.edu/ (visited on 2019-06-22).

Balai, Evgenii and Michael Gelfond (2016). "On the Relationship between P-log and LPMLN." In: *IJCAI'16 Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. AAAI Press, pp. 915–921.

Baral, Chitta, Michael Gelfond, and Nelson Rushton (2009). "Probabilistic Reasoning with Answer Sets". In: *Theory and Practice of Logic Programming* 9, pp. 57–144.

Bishop, Christopher M (2006). *Pattern recognition and machine learning*. Springer.

Brewka, Gerhard, Thomas Eiter, and Mirosław Truszczyński (2011). "Answer set programming at a glance". In: *Communications of the ACM* 54.12, pp. 92–103.

Buccafurri, Francesco, Nicola Leone, and Pasquale Rullo (2000). "Enhancing disjunctive datalog by constraints". In: *IEEE Transactions on Knowledge and Data Engineering* 12.5, pp. 845–860.

Cabalar, Pedro (2009). "Existential Quantifiers in the Rule Body". In: *Proceedings of the 23rd Workshop on (Constraint) Logic Programming 2009*. Ed. by Ulrich Geske and Armin Wolf. Potsdam.

Calimeri, Francesco et al. (2012). "ASP-Core-2: Input language format". In: *ASP Standardization Working Group, Technical report*.

Eiter, Thomas and Tobias Kaminski (2016). "Exploiting Contextual Knowledge for Hybrid Classification of Visual Objects". In: *European Conference on Logics in Artificial Intelligence*. Springer, pp. 223–239.

Eiter, Thomas, Mustafa Mehuljic, et al. (2015). *DLVHEX: User Guide*.

Everardo, Flavio, Roland Kaminski, and Marius Lindauer (2019). *xorro*. URL: https://github.com/potassco/xorro (visited on 2019-11-04).

Ferraris, Paolo, Joohyung Lee, and Vladimir Lifschitz (2011). "Stable models and circumscription". In: *Artificial Intelligence* 175.1, pp. 236–263.

Fierens, Daan et al. (2015). "Inference and learning in probabilistic logic programs using weighted boolean formulas". In: *Theory and Practice of Logic Programming* 15.3, pp. 358–401.

Gebser, M. et al. (2017). *Potasco: User Guide*. https://github.com/potassco/guide/releases/download/v2.1.0/guide.pdf. Accessed 16-11-2018.

Gelfond, Michael and Vladimir Lifschitz (1988). "The stable model semantics for logic programming." In: *ICLP/SLP*. Vol. 88, pp. 1070–1080.

Gelfond, Michael and Vladimir Lifschitz (1991). "Classical negation in logic programs and disjunctive databases". In: *New generation computing* 9.3-4, pp. 365–385.

Getoor, Lise and Ben Taskar, eds. (2007). *Introduction to Statistical Relational Learning*. London: MIT Press.

Kautz, Henry A, Bart Selman, and Yueyen Jiang (1996). "A general stochastic approach to solving problems with hard and soft constraints." In: *Satisfiability Problem: Theory and Applications* 35, pp. 573–586.

Kimmig, Angelika, Lilyana Mihalkova, and Lise Getoor (2015). "Lifted graphical models: a survey". In: *Machine Learning* 99.1, pp. 1–45.

Koller, Daphne and Nir Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques*. Cambridge: The MIT Press.

Lee, Joohyung and Vladimir Lifschitz (2003). "Loop Formulas for Disjunctive Logic Programs". In: *Nineteenth Internationcal Conference on Logic Programming*, pp. 451–465.

Lee, Joohyung and Yunsong Meng (2011). "First-order stable model semantics and first-order loop formulas". In: *Journal of Artificial Intelligence Research* 42, pp. 125–180.

Lee, Joohyung, Yunsong Meng, and Yi Wang (2015). "Markov Logic Style Weighted Rules under the Stable Model Semantics." In:

Lee, Joohyung, Samidh Talsania, and Yi Wang (2017a). *LP^{MLN}-sys manual*. Version Version 2.2.x. 71 pp. URL: `http://reasoning.eas.asu.edu/lpmln/manual/sysmanual.pdf` (visited on 2019-09-21).

— (2017b). "Computing LP MLN using ASP and MLN solvers". In: *Theory and Practice of Logic Programming* 17.5-6, pp. 942–960.

— (2019). *LP^{MLN}*. URL: `http://reasoning.eas.asu.edu/lpmln/` (visited on 2019-11-04).

Lee, Joohyung and Yi Wang (2016a). "Weighted Rules under the Stable Model Semantics". In: *Fifteenth Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, pp. 145–154.

— (2016b). *Weighted Rules under the Stable Model Semantics*. `http://reasoning.eas.asu.edu/papers/lpmln-kr-long.pdf`. Accessed: 30-09-2018.

— (2018). "Weight learning in a probabilistic extension of answer set programs". In: *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 22–31.

Lee, Joohyung and Zhun Yang (2017). "LPMLN, Weak Constraints, and P-log". In: *Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 1170–1177.

Lin, Fangzhen and Yuting Zhao (2004). "ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers". In: *Artificial Intelligence* 157, pp. 115–137.

Lowd, Daniel and Pedro Domingos (2007). "Efficient Weight Learning for Markov Logic Networks". In: *Knowledge Discovery in Databases: PKDD 2007*. Ed. by J.N. et al Kok. Berlin: Springer, pp. 200–211.

LPMLN learning system (2019). URL: `https://lpmln-learn.weebly.com/` (visited on 2019-06-22).

Niu, Feng et al. (2011). “Tuffy: Scaling up statistical inference in markov logic networks using an rdbms”. In: *Proceedings of the VLDB Endowment* 4.6, pp. 373–384.

Pedregosa, F. et al. (2011). “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12, pp. 2825–2830. URL: https://scikit-learn.org/stable/modules/classes.html (visited on 2019-12-14).

Poon, Hoifung and Pedro Domingos (2006). “Sound and efficient inference with probabilistic and deterministic dependencies”. In: *AAAI*. Vol. 6. AAAI Press, pp. 458–463.

ProbLog 2.1 documentation (2014). Online. Leuven. URL: https://problog.readthedocs.io/en/latest/index.html#.

Richardson, Matthew and Pedro Domingos (2006). “Markov Logic Networks”. In: *Machine Learning* 62, pp. 107–136.

Russell, Bryan C et al. (2008). “LabelMe: a database and web-based tool for image annotation”. In: *International journal of computer vision* 77.1-3, pp. 157–173.

Sato, Taisuke (1995). “A statistical learning method for logic programs with distribution semantics”. In: *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*. Citeseer. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.4408.

Sen, Prithviraj et al. (2010a). “Collective Classification”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, pp. 189–193. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_140. URL: https://doi.org/10.1007/978-0-387-30164-8_140.

— (2010b). “Collective Classification”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Springer.

Singla, Parag and Pedro Domingos (2005). “Discriminative Training of Markov Logic Networks”. In: *American Association for Artificial Intelligence* 5, pp. 868–873.

Van Gelder, Allen, Kenneth A Ross, and John S Schlipf (1991). “The well-founded semantics for general logic programs”. In: *Journal of the ACM (JACM)* 38.3, pp. 619–649.

Van Haaren, Jan et al. (2016). “Lifted generative learning of Markov logic networks”. eng. In: *Machine Learning* 103.1, pp. 27–55. ISSN: 0885-6125.

Wang, Bin et al. (2018). “Splitting an LPMLN Program”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*.

Wei, Wei, Jordan Erenrich, and Bart Selman (2004). “Towards efficient sampling: Exploiting random walk strategies”. In: *AAAI* 4, pp. 670–676.