FoxPSL: An Extended and Scalable PSL Implementation

Sara Magliacane
VU University Amsterdam
De Boelelaan 1081a, 1081HV
Amsterdam, The Netherlands
s.magliacane@vu.nl

Philip Stutz
University of Zurich
Binzmuhlestrasse 14, 8050
Zurich, Switzerland
stutz@ifi.uzh.ch

Paul Groth
VU University Amsterdam
De Boelelaan 1081a, 1081HV
Amsterdam, The Netherlands
p.t.groth@vu.nl

Abraham Bernstein
University of Zurich
Binzmuhlestrasse 14, 8050
Zurich, Switzerland
bernstein@ifi.uzh.ch

Abstract

In this paper we present *foxPSL*, an extended and scalable implementation of Probabilistic Soft Logic (PSL) based on the distributed graph processing framework SIGNAL/COLLECT. PSL is a template language for hinge-loss Markov Random Fields, in which MAP inference is formulated as a constrained convex minimization problem. A key feature of PSL is the capability to represent soft truth values, allowing the expression of complex domain knowledge.

To the best of our knowledge, *foxPSL* is the first end-to-end distributed PSL implementation, supporting the full PSL pipeline from problem definition to a distributed solver that implements the Alternating Direction Method of Multipliers (ADMM) consensus optimization. *foxPSL* provides a Domain Specific Language that extends standard PSL with a type system and existential quantifiers, allowing for efficient grounding. We compare the performance of *foxPSL* to a state-of-the-art implementation of ADMM consensus optimization in GraphLab, and show that *foxPSL* improves both inference time and solution quality.

Introduction

Probabilistic Soft Logic (PSL) (Broecheler, Mihalkova, and Getoor 2010; Bach et al. 2012; 2013) is a template language for hinge-loss Markov random fields. Similar to other statistical relational learning formalisms like Markov Logic Networks (MLN), in PSL the first order logic formulae representing the templates can be instantiated (grounded) using the individuals in the domain, creating a Markov random field on which one can perform inference tasks.

A key feature of PSL is the capability to represent and combine soft truth values, i.e. truth values in the interval [0,1], allowing the expression of degrees of relationships within complex domain knowledge, such as the degree of relationships. Given the continuous nature of the truth values, the use of Lukasiewicz operators and the restriction of logical formulae to Horn clauses with disjunctive heads, Maximum a posteriori (MAP) inference in PSL can be formulated as a constrained convex minimization problem. This problem can be cast as a consensus optimization problem (Bach et al. 2012; 2013) and solved very efficiently with distributed algorithms like the Alternating Direction Method of Multipliers (ADMM), recently popularized by (Boyd et al. 2011).

The current reference implementation of PSL, described in (Bach et al. 2012; 2013), is limited to running on one machine, which limits the solvable problem sizes. In (Miao et al. 2013), PSL is used as a motivation for the development of ACO, a vertex programming algorithm for ADMM consensus optimization that works in a distributed environment. In that work, GraphLab (Low et al. 2010) is used as the underlying processing framework. Here, we build on the insights of (Miao et al. 2013), but instead of focusing on the general problem of ADMM distributed consensus optimization, we focus on designing an environment specifically for supporting PSL. The result is *foxPSL*, to the best of our knowledge the first end-to-end distributed implementation of PSL that provides an environment for working with large PSL domains. Our evaluations show that *foxPSL*'s ADMM implementation is faster than ACO while offering an array of additional features for the expression of PSL domains.

Like (Miao et al. 2013), we adopt distributed graph processing for the basis of our implementation. Instead of GraphLab, we implement ADMM consensus optimization in SIGNAL/COLLECT (Stutz, Bernstein, and Cohen 2010), which enables a natural and more extensible representation of ADMM graphs. Furthermore, we provide a domain specific language that extends PSL with a type system, partially grounded rules and existential quantification. Using these additional language features we can optimize the translation of PSL domains to ADMM, for example by providing type-aware existential quantification and grounding.

Summarizing, our contributions are (i) a publicly released end-to-end PSL environment[1] that (ii) supports a broad range of extended PSL features and optimizations, (iii) seamlessly parallelizes and distributes computations and consistently outperforms the state of the art both in inference time and solution quality. In the following we describe *foxPSL* and its features, present an empirical evaluation, and conclude with a discussion of future work.

System Description

foxPSL is designed as a pipeline layered on top of the SIGNAL/COLLECT graph processing framework. In this section

[1] Apache 2.0 licensed, `https://github.com/uzh/fox`

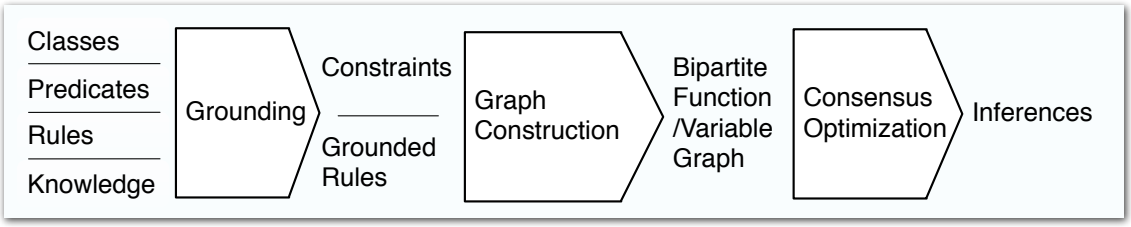


Figure 1: The architecture of our PSL inferencing system is a pipeline that takes the rules, instance knowledge and metadata as inputs and through a series of processing steps computes the inferences.

we describe the underlying framework and each stage of the pipeline (see Figure 1) with its inputs and outputs.

Underlying Graph Processing in Signal/Collect SIGNAL/COLLECT (Stutz, Bernstein, and Cohen 2010)[2] is a parallel and distributed graph processing framework. Akin to GraphLab (Gonzalez et al. 2012), it allows to formulate computations in terms of vertex centric methods. In SIGNAL/COLLECT functions are separated into ones for aggregating received messages (collecting) and ones for computing messages sent along edges (signalling). In contrast to GraphLab, SIGNAL/COLLECT supports different vertex types for different processing tasks, which allows for a natural representation of bipartite ADMM consensus graphs. Moreover, it provides configurable convergence detection based on local or global properties.

PSL Input via *foxPSL* language The input to the system is a description of individuals, predicates, facts and rules in a domain-specific language for PSL. In the following, we comment an example from the repository[3].

In *foxPSL*, it is possible to explicitly list individuals in our domain, and optionally assign them to a class. For example:

```
class Person: anna, bob
class Party:  demo, repub
class Woman:  anna
individuals:  ufo
```

By convention individuals always start with a lower-case letter. Our domain consists of two individuals of class Person ($anna, bob$), two of class Party ($demo$, $repub$), one of class Woman ($anna$) and one individual without any class (ufo). Classes are not mutually exclusive and the same individual can have multiple classes ($anna$). Besides explicitly mentioned individuals, *foxPSL* can automatically infer other individuals and their class from facts and rules.

For each predicate, we can specify the classes of its arguments, enabling a more efficient grounding:

```
predicate: retired(_)
predicate: professor(Person)
```

In the example, the predicate $retired$ takes one argument of any class, while $professor$ takes one argument of class Person. In the grounding, the only individuals used to ground $professor$ will be those of class Person, greatly reducing the number of grounded predicates.

As in standard PSL, we can define predicate properties like functionality, partial functionality and symmetry, which are translated into constraints on grounded predicates.

```
predicate [Functional]:  votes(Person,Party)
predicate[Symmetric]: friends(Person,Person)
```

In the example, the functionality of $votes$ means that the votes for different parties that a person can cast must sum up to 1. The symmetry of $married$ means that for all individuals a, b, if $married(a, b) = x$ then $married(b, a) = x$.

Once we defined the predicates, we can state some facts about our domain and their truth values.

```
fact [truthValue = 0.8]: friends(bob, carl)
fact [truthValue = 0.9]: !votes(anna, demo)
```

In our domain, bob is a friend of $carl$ with truth value 0.8. Although $carl$ was not mentioned as a Person before, this is inferred from the $friends$ fact. Moreover, $anna$ does not vote for $demo$ with truth value 0.9, i.e. $votes(anna, demo) = 0.1$.

The core of *foxPSL* is the definition of rules in the form :

$$B_1 \wedge ... \wedge B_n \Rightarrow H_1 \vee ... \vee H_m$$

where H_i for $i = 1, ..., m$ and B_j for $j = 1, ..., n$ are literals. The restriction to this form is a constraint of standard PSL that enables the translation to convex functions when considering Lukasiewicz operators. For example:

```
rule [5]: votes(A,P) & friends(A,B) =>
          votes(B,P)
rule [3, linear]: young(P) =>  !retired(P)
rule: professor(P) => EXISTS [C,S]
        teaches(P,C,S) | retired(P)
```

Similar to standard PSL, each rule can have an associated weight that represents how strict it is and associated distance measure (e.g. linear, squared) that describes the shape of the penalty function for breaking this rule.

In addition to standard PSL, we introduce the existential quantifier $EXISTS[variables]$, which can only appear in the head, in order to preserve convexity. If the weight is not specified, we consider the rule a hard rule, i.e. a constraint.

[2] http://uzh.github.io/signal-collect/
[3] http://tinyurl.com/FoxPSLExample

Grounding For each rule we instantiate each argument in each predicate with all suitable individuals, creating several grounded predicates from a single predicate. For example $votes(A, P)$ from the first rule produces 6 grounded predicates: $votes(anna, demo)$, $votes(bob, demo)$, etc. Some of these grounded predicates are user provided facts with truth values (e.g. $votes(anna, demo) = 0.1$), but most of them have an unknown truth value that we will compute.

We substitute the grounded predicates in all combinations in each rule, generating several grounded rules. The same substitution is done for all constraints. For example, the first rule generates 18 grounded rules, e.g. $votes(anna, demo)$ $\&\&$ $friends(anna, bob) => votes(bob, demo)$. We also ground the existential quantifiers by unrolling them to disjunctions of grounded predicates with the matching classes.

Graph Construction The grounding step produces a set of grounded rules and constraints, each containing multiple instances of grounded predicates. As defined in PSL (Bach et al. 2012), each grounded rule or constraint is translated to the corresponding potential of a continuous Markov random field using Lukasiewicz operators. In particular, for a grounded rule:

$$[weight]\ b_1 \wedge ... \wedge b_n \Rightarrow h_1 \vee ... \vee h_m$$

we can define the distance to satisfaction:

$$weight * max(0, b_1 + ... + b_n - n + 1 - h_1 - ... - h_m)^p$$

where $p = 1$ for rules with linear distance measures and $p = 2$ for rules with squared distance measures. Since the potentials are convex functions and the constraints are defined to be linear functions, MAP inference becomes a constrained convex minimization problem. Following the approach described in (Bach et al. 2013; Miao et al. 2013) we solve the MAP inference using consensus optimization. We represent the problem as a bipartite graph, where grounded rules and constraints are represented with function (also called subproblem) vertices and grounded predicates are represented with consensus variable vertices. Each function (grounded rule or constraint) has a bidirectional edge pointing to every consensus variable (grounded predicate) it contains.

Consensus Optimization with ADMM The function and consensus variable vertices implement the ADMM consensus optimization (Boyd et al. 2011). Intuitively, in each iteration each function is minimized separately based on the consensus assignment from the last iteration and the related Lagrange multipliers. Once done, the function vertex sends a message with the preferred variable assignment to all connected consensus variables. Each consensus variable computes a new consensus assignment based on the values from all connected functions, for example by averaging, and sends it back. This process is repeated until convergence, or in the approximate case, until the primal and dual residuals fall below a threshold based on the parameters $\epsilon_{abs}, \epsilon_{rel}$.

Since each variable represents a grounded predicate, the assignment to a variable is the inferred soft truth value for that grounded predicate. At the end of the computation, the system outputs all the inferred truth values.

Evaluation

We compare *foxPSL* with ACO, the GraphLab ADMM consensus optimization implementation presented in (Miao et al. 2013), measuring both the inference time and solution quality. The comparison is based on the datasets used in (Miao et al. 2013), which represent four synthetic social networks of increasing size modelling voter behaviour. The datasets contain from 4.41 to 17.7 million vertices. All evaluations are run on four machines, each with 128 GB RAM and two E5-2680 v2 at 2.80GHz 10-core processors. All machines are connected with 40Gbps Infiniband.

Both systems run an approximate version of the ADMM algorithm, as described in (Boyd et al. 2011), with the same parameters $\rho = 1$, $\epsilon_{abs} = 10^{-5}$ and $\epsilon_{rel} = 10^{-3}$. The main difference is in the detection algorithm: while ACO implements a special convergence detection that computes a local approximation of the residuals in each consensus vertex, *foxPSL* employs the textbook global convergence detection, efficiently implemented as a MapReduce aggregation. The ACO convergence detection reduces the computation in parts of the graph that have almost converged at the cost of a coarser approximated solution. On the other hand, in certain cases, some local computations stop too soon, requiring others to run for several iterations to compensate.

Inference Time Comparison

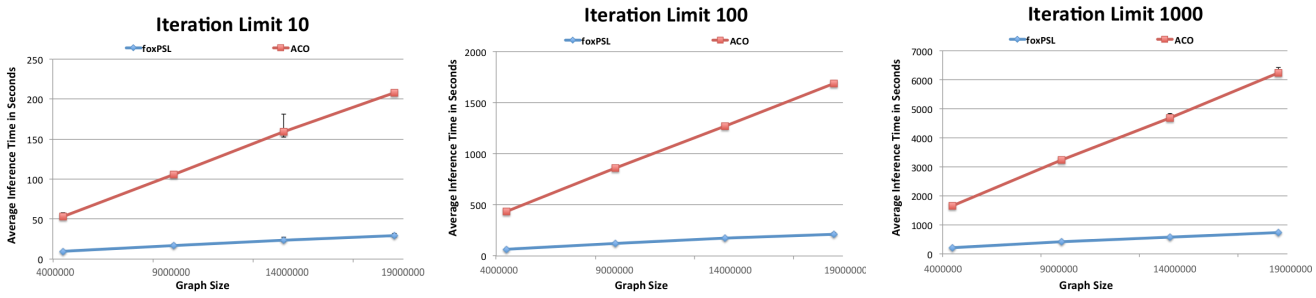
For each of the four datasets, we measure the inference time at a fixed number of iterations for both systems. Figure 2 shows the results averaged over ten runs, limited to 10, 100, and 1000 iterations. Since the ACO implementation performs two extra initial iterations that are not counted in the limit, the comparison is made with an iteration limit of 12, 102, and 1002 for *foxPSL*. We stop the evaluation at 1000 iterations, because *foxPSL* converges in that interval, although ACO does not, running for several more iterations without substantial improvements in quality.

The inference time of *foxPSL* is considerably better in all considered iterations, as shown by the lower computation times (y-axis) in Figures 2. Moreover, the computation time seems to scale linearly with increasing dataset sizes (x-axis).

Solution Quality Comparison

We also compare *foxPSL* and ACO in terms of the solution quality on the same evaluation runs discussed above. Since PSL inference is a constrained minimization problem solved with an approximate algorithm, we consider two quality measures for solution quality: the objective function value and a measure of the violation of the constraints.

In Table 1, we show a comparison of the solution quality for *foxPSL* and ACO at iteration 1000 from the previous experiments. The four parameters we compare are the objective function value, the sum of violations of the constraints, and the number of violated constraints at tolerance level 0.01 and at tolerance level 0.1. We estimate the optimal objective function value as computed using lower epsilons ($\epsilon_{abs} = 10^{-8}$, $\epsilon_{rel} = 10^{-8}$) with a sum of constraint violations in the order of 10^{-5}. We show that *foxPSL*'s objective function value is closer to the estimated optimal value than



(a) (b) (c)

Figure 2: Figures 2(a), 2(b) and 2(c), compare the average inference time between *foxPSL* and ACO. For each graph size there were ten runs, with an iteration limit of 10, 100 and 1000 respectively. The error bars indicate the fastest and slowest runs.

		foxPSL				ACO			
$\|vertices\|$	$OptVal$	$ObjVal$	$\Sigma viol$	$\|viol@0.01\|$	$\|viol@0.1\|$	$ObjVal$	$\Sigma viol$	$\|viol@0.01\|$	$\|viol@0.1\|$
4.41M	4838.14	4810.36	7.77	20	4	4722.09	119.55	491	233
8.86M	9692.03	9679.13	13.82	18	3	9520.12	233.04	924	431
13.2M	14521.26	14449.47	21.14	42	6	14199.59	349.14	1387	640
17.7M	19425.71	19441.69	27.12	52	5	19119.38	472.49	1894	863

Table 1: Comparison of solution quality for *foxPSL* and ACO with iteration limit 100.

ACO's for all datasets. The sum of the violations is lower in *foxPSL* by one order of magnitude.

Besides the lower total violation of the constraints, Table 1 also shows the number of constraints that are violated by more than a certain threshold of tolerance. In particular, in the ACO solution there are several hundred constraints that are violated even while tolerating errors of 0.1, which is sizeable considering that the variables are constrained in the interval [0, 1]. In general, the violations of the ACO solution are larger and less spread across the constraints than the ones found in the *foxPSL* solution, possibly due to the local convergence detection of the former, which may be too eager to stop computation on some subgraphs.

Limitations and Conclusions

In this paper, we introduced *foxPSL*, to our knowledge the first end-to-end PSL implementation that provides a DSL for specifying the problem definition and enables scaling via optimized grounding and distributed processing.

Our current implementation has limitations. Our ADMM algorithm uses a fixed step size, leading to an initially fast approximation of the result and a slow exact convergence. An improvement would be a variant with adaptive step sizes. An additional improvement might be the use of asynchronous execution and incremental reasoning when facts/rules change – both endeavours we leave for future work.

Whilst developing *foxPSL*, we found that PSL provides a powerful formalism for modelling problem domains. However, its power comes with numerous interactions between its elements. Hence, the use of PSL would be immensely aided by a DSL and end-to-end environment that allows to systematically analyse these interactions. We believe that *foxPSL* is a first step towards such an environment.

Acknowledgements We would like to thank Hui Miao for sharing the ACO code and evaluation datasets, and Stephen Bach for being able to use his optimizer implementations in *foxPSL*. We also thank the Hasler Foundation and Dutch national programme COMMIT for supporting this work.

References

Bach, S. H.; Broecheler, M.; Getoor, L.; and O'Leary, D. P. 2012. Scaling MPE inference for constrained continuous Markov random fields. In *NIPS 2012*.

Bach, S. H.; Huang, B.; London, B.; and Getoor, L. 2013. Hinge-loss Markov random fields: Convex inference for structured prediction. In *UAI 2013*.

Boyd, S.; Parikh, N.; Chu, E.; Peleato, B.; and Eckstein, J. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.* 3(1).

Broecheler, M.; Mihalkova, L.; and Getoor, L. 2010. Probabilistic similarity logic. In *UAI 2010*.

Gonzalez, J. E.; Low, Y.; Gu, H.; Bickson, D.; and Guestrin, C. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI 2012*.

Low, Y.; Gonzalez, J.; Kyrola, A.; Bickson, D.; Guestrin, C.; and Hellerstein, J. M. 2010. Graphlab: A new parallel framework for machine learning. In *UAI 2010*.

Miao, H.; Liu, X.; Huang, B.; and Getoor, L. 2013. A hypergraph-partitioned vertex programming approach for large-scale consensus optimization. In *IEEE Big Data 2013*.

Stutz, P.; Bernstein, A.; and Cohen, W. W. 2010. Signal/-Collect: Graph Algorithms for the (Semantic) Web. In *ISWC 2010*.