Declarative Probabilistic Programming for Undirected Graphical Models: Open Up to Scale Up

Sebastian Riedel
Department of Computer Science, University of Massachusetts Amherst, Amherst, MA 01002
riedel@cs.umass.edu

Abstract

We argue that probabilistic programming with undirected models, in order to scale up, needs to *open up*. That is, instead of focusing on minimal sets of generic building blocks such as universal quantification or logical connectives, languages should *grow* to include specific building blocks for as many uses cases as necessary. This can not only lead to more concise models, but also to *more efficient* inference if we use methods that can exploit building-block specific sub-routines. As embodiment of this paradigm we present FAST FROWARD, a platform for implementing probabilistic programming languages that grow.

Introduction

Probabilistic Programming languages for undirected models, such as Markov Logic and Relational Markov Networks, are very expressive. Many statistical models of interest can be readily described in terms of these languages. However, often the generic inference routines will either be too slow, too inaccurate, or both. For example, it is possible to use Markov Logic (Richardson and Domingos 2006) for encoding probability distributions over the set of possible syntactic dependency trees of a sentence. Yet, generic inference in these models is very inefficient, in particular due to deterministic factors which ensure that the set of predicted edges forms a spanning tree over the words of the sentence.

Here we argue that probabilistic programming, in order to scale up, needs to *open up*. That is, instead of focusing on minimal sets of generic building blocks such as universal quantification and logical connectives, languages should *grow* to eventually include specific building blocks for as many uses cases as necessary. For example, we should provide a spanning tree constraint as part of our language that can be used whenever we want to extract dependency trees, or model hierarchical structures in general.

On first sight, this is not more than syntactic sugar. However, we argue that it can also lead to *more efficient* inference if we use inference methods that can exploit building-block specific sub-routines. For example, Belief Propagation requires summation over the variables of each factor. For a spanning tree constraint this can be done very efficiently.

Clearly, we do not want to design a language with all constructs we could ever need in advance. Instead, we need to provide the glue for an ever-increasing set of language extensions. In the following we will present FAST FROWARD,[1] a software library that attempts to provide this glue.

FAST FROWARD supports user-provided inference routines akin to the (primarily) imperative language FACTORIE (McCallum, Schultz, and Singh 2009), but does so in a declarative setting. It is also similar in spirit to recent approaches such as CHURCH (Goodman et al. 2008) and FIGARO (Pfeffer 2009), which support tailor-made proposal functions. However, these languages focus on generative models and MCMC.

Domains, Variables, Worlds

The glue that FAST FROWARD provides are object-oriented interfaces for building blocks, and high level inference routines that function in terms of these interfaces. We present the former in Scala, a hybrid functional object-oriented programming language.

A `Domain[T]` contains the (Scala) objects of type `T` we want to talk about; it needs to provide an iterator over its objects, as well as a `contains` method to indicate Domain membership. Three built-in types of domains are `Values`, representing a user-defined set of objects, `Tuples`, and `Functions`. For example, in

```
val Tokens = Values(0,1,2,3,4,5)
val Graph = (Tokens x Tokens) -> Bools
```

the first domain `Tokens` contains all integers from 0 to 5 (and represent word indices in a sentence), and the second domain all functions that map token pairs to booleans (and represent directed graphs over the tokens).

In FAST FROWARD a variable is represented by objects of the class `Var[T]` that come with a name and a domain that specifies which values the variable can

[1] `http://riedelcastro.github.com/fast-froward/`.

possibly take on. For example, to declare the variable `edge` as a graph over tokens we write `val edge = Var("edge", Graph)`. A binding of such variables is a (possible) `World`.[2] Its core method `resolveVar(v)` returns the object the variable `v` is assigned to.

Terms

A term is a symbolic expression that is, given a possible world, evaluated to an object. In FAST FROWARD a term is an instance of a `Term[T]` trait that has to implement an `evaluate(world)` method which maps the binding `world` to a value of type `T`. Terms in FAST FROWARD can serve as boolean formulae (when they evaluate into booleans). Crucially, when they evaluate into real values they also serve as probabilistic models.

A simple example term is:

```
Indicator(FunApp(edge,Tuple(0,5))
```

`Indicator` evaluates to 1 if the boolean term inside evaluates to `TRUE`, and 0 otherwise. The `FunApp` term applies the result of evaluating the first argument to the result of evaluating the second. `edge` refers to the `Var` object we defined earlier and evaluates to the function the variable is bound to. The other terms are defined accordingly. Scala's syntactic sugar, and `${·}` as the indicator function, can be used to alternatively write `${edge(0,5)}`.

To support abstraction, we provide quantified operations that are applied to all objects of a given domain. For example, to evaluate

```
Sum(Tokens,i=>${tag(i,VB)->edge(0,i)})
```

we replace i in the inner term with each possible value in `Tokens`, then we evaluate the inner term, get a real value x_i, and sum over all values x_i. Note that the term amounts to the logarithm of a unnormalized MLN that encourages worlds where verbs are children of the root (by convention token 0). Generic MLNs can be formulated accordingly.

Crucially, we can also add terms that are not in Markov Logic. For example: the spanning tree constraint `Tree(edge)` for which `eval` returns 1 if the function in `edge` corresponds to a spanning tree over objects in `Tokens`, and 0 otherwise.

Inference

Real valued terms also implement a `factorize` method that returns a set of terms it factors into. This method, together with `eval`, is sufficient to implement most (propositional) factor graph inference methods such as Sum Product Belief Propagation and its variants.

However, working purely in terms of `eval` will often be very inefficient. For example, calculating outgoing BP messages for `Tree(edge)` in this way is intractable because `eval` needs to be called for each of the exponential assignments to `edge`. We therefore introduce a `bpMessage` method that terms implement if outgoing messages can be calculated more efficiently. For our tree constraint factor this is possible in cubic time (Smith and Eisner 2008).

There is a more general paradigm behind inference in FAST FROWARD: the probabilistic programmer composes a probabilistic model, and the interpreter composes a global optimizer for this model using the local optimizers that come with its building blocks. This technique is appropriate not only for Sum-Product BP, but also for other methods that break up the global variational objective into several local problems via dual decomposition (Duchi et al. 2007; Komodakis, Paragios, and Tziritas 2007).

Note that a similar approach can also be used for methods that do not unroll the full graph. For example, quantified terms can have a `separate` method for Cutting Plane Inference (Riedel 2008) that returns all factors not *maximally satisfied* in a given world.

Conclusion

We have presented FAST FROWARD, an object-oriented library for probabilistic programming languages that grow. Crucially, language extensions can come with specialized inference routines that FAST FROWARD can leverage. We believe that this approach will make probabilistic programming with undirected models more applicable to real-world problems.

Acknowledgments This work was supported in part by the CIIR, in part by SRI International subcontract #27-001338 and ARFL prime contract #FA8750-09-C-0181, and in part by UPenn NSF medium IIS-0803847. Any opinions, findings and conclusions or recommendations expressed in this material are the author's and do not necessarily reflect those of the sponsor.

References

Duchi, J.; Tarlow, D.; Elidan, G.; and Koller, D. 2007. Using combinatorial optimization within max-product belief propagation. In *NIPS '07)*.

Goodman, N. D.; Mansinghka, V. K.; Roy, D.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *UAI '08*.

Komodakis, N.; Paragios, N.; and Tziritas, G. 2007. Mrf optimization via dual decomposition: Message-passing revisited. In *In ICCV '07*.

McCallum, A.; Schultz, K.; and Singh, S. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS '09)*.

Pfeffer, A. 2009. Figaro: An object-oriented probabilistic programming language. In *Charles River Analytics Technical Report (2009)*.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62:107–136.

Riedel, S. 2008. Improving the accuracy and efficiency of MAP inference for markov logic. In *UAI '08*.

Smith, D. A., and Eisner, J. 2008. Dependency parsing by belief propagation. In *EMNLP '08*.

[2] This amounts to possible worlds in Markov Logic for bindings of function variables that map into booleans.