Practical Probabilistic Programming with Figaro

Avi Pfeffer
Charles River Analytics
apfeffer@cra.com

Brian Ruttenberg
Charles River Analytics
bruttenberg@cra.com

Michael Howard
Charles River Analytics
mhoward@cra.com

Glenn Takata
Charles River Analytics
gtakata@cra.com

Joe Gorman
Charles River Analytics
jgorman@cra.com

Alison O'Connor
Charles River Analytics
aoconnor@cra.com

Abstract

Figaro is an object–oriented, functional probabilistic programming language (PPL). As an embedded library within Scala, Figaro is a flexible, modular, and powerful PPL that enables users to construct a wide variety of rich, complex, and relational models in a general purpose programming language. Coupled with diverse suite of built-in inference algorithms, Figaro provides the tools needed for users to build practical and real–world artificial intelligence applications.

1 Motivation

Probabilistic reasoning systems are used for a number of purposes, including predicting future events, inferring past causes of current observations, and learning from observations to better predict in the future. There is a wide variety of existing probabilistic representations, such as Bayesian networks [1] , Markov networks [2], probabilistic relational models [3] , Markov logic [4], and probabilistic grammars [5]. While each of these languages is good at representing what it can represent, they are all constrained in scope and limited to a certain type of model, and it would be hard to develop a sophisticated model of a full, complex system with them.

On the other hand, probabilistic languages have existed for a long time that model complex systems of this type, namely simulation languages. Simulations have been used to model sports seasons or military campaigns in great detail. The problem with simulations, however, is that they can only be used to predict what will happen. They cant be used to analyze the causes of what actually happened, and the ability to learn from what actually happened to predict better in the future is limited.

Essentially, probabilistic programming tries to bring together the best of traditional probabilistic reasoning systems, with their flexible inferencing and learning capabilities, with the expressive power of simulation languages. We have been involved in probabilistic programming since its early days [6]. We developed IBAL [7], one of the first general-purpose programming systems. Currently, we are developing Figaro [8], which is maintained open-source on Github and is currently on version 3.3.

The design of Figaro has been guided by our experience with IBAL. IBAL was a highly expressive, Turing complete language (albeit only with discrete variables), and had sophisticated exact and approximate inference algorithms. However, working with IBAL, we found it difficult to create applications, because the facilities to interact with data and integrate with other applications were limited. Therefore, our main goal with Figaro has been to develop a mature probabilistic programming system that can be the basis for powerful applications. At the same time, it was vital not to sacrifice any of the expressive power of IBAL and even increase it by providing support for continuous variables and undirected models. Also, we have developed a rich and constantly growing set of inference algorithms for Figaro including variable elimination, belief propagation, Markov chain Monte Carlo, importance sampling, particle filtering, decision making, EM–based learning, and more.

2 Figaro Design

Motivated by the need for practicality, the key feature of Figaro is that it is an embedded library in Scala. This distinguishes it from languages like IBAL, BLOG [9], and Church [10], which are their own language rather than embedded in another language. Other languages, such as Infer.NET [11], have also taken the approach of being embedded, but these languages are generally much less expressive than Figaro. The corollary of being embedded in Scala is that any Scala function can be included in a Figaro model, which is an extremely powerful feature.

Since Figaro is embedded in Scala, it is more properly viewed as a library of data structures for representing probabilistic models than purely a language. These data structures are centered around the probability monad [12]. Instances of these data structures are known as "elements". There are atomic elements, such as Flip, Binomial, and Normal. There is an element called Chain, which represents the monadic bind operation. Chain works like a node in a Bayesian network, which gets the value of the parents, and based on that value chooses a distribution over the child. Its name is derived from the chain rule of probability, and it serves as the glue when chaining a sequence of variables together into a model. Given a parent value, the choice of which distribution to use for the child is defined by a Scala function; this is one way in which Scala functions appear in Figaro models. For example,

```
Chain(Uniform(0,1), (d: Double) => Normal(d, 0.5))
```

defines a process that first generates a uniformly distributed number between 0 and 1, and then applies the Scala function that takes this number and produces a normal distribution with that number as the mean.

Another way Scala functions can appear in Figaro programs is with the Apply element, which serves to lift functions on values to elements. For example, `Apply(Normal(0, 1), (d: Double) => d * 2)` defines the process that first generates a normally distributed number and then multiplies it by 2. Many other Figaro constructs are syntactic sugar.

Two other salient features of Figaros design are its constraint system and its names and references. Any element can have constraints, which are functions from values of the element to real numbers. Their effect is like a potential in a Markov network. Since we can group together multiple elements into a single element using Apply, this gives Figaro the ability to express undirected models.

Names and references are useful for object-oriented modeling, for example using probabilistic relational models. Every element belongs to an element collection and may have a name that uniquely identifies it in its element collection. The value of an element may itself be an element collection this provides the ability to string names together to refer to an embedded element. Of course, there may be uncertainty over which element a reference refers to, which is handled seamlessly by Figaro. We have also found references to be useful in hierarchical taxonomic reasoning and in dynamic models.

3 Examples

Figaro has been used in applications such as space object identification, target tracking, malware analysis, and soil drainage prediction. While Figaros power is most evident in complex applications, we illustrate its main features with three simple examples. The first example shows the use of constraints to define an undirected model.

```
class Person { val smokes = Flip(0.6) }
val alice, bob, clara = new Person
val friends = List((alice, bob), (bob, clara))
```

```
4  clara.smokes.observe(true)
5  def smokingInfluence(pair: (Boolean, Boolean)) =
6    if (pair._1 == pair._2) 3.0; else 1.0
7  for { (p1, p2) <- friends } {
8    ^^(p1.smokes, p2.smokes).setConstraint(smokingInfluence)
9  }
```

This is the classic friends and smokers example from Markov logic. Line 1 defines a Person class with a smokes element that is true with probability 0.6. Line 2 creates three instances of Person, while line 3 says which pairs are friends and line 4 asserts that one of them smokes. Although these have been defined inline for the purpose of example, the people, friends, and observations could easily be read from an external database. Lines 5-6 defines a constraint that asserts that two peoples smoking habits are 3 times as likely to be the same than different, while lines 7 to 10 apply this constraint to all pairs of friends. Using Metropolis-Hastings with 20,000 samples, Figaro returns that the probability of Alice smoking is 0.73625. The next example illustrates object-oriented reasoning using references.

```
1  abstract class Engine extends ElementCollection {
2    val power: Element[Symbol]
3  }
4  class V8 extends Engine {
5    val power: Element[Symbol] =
6      Select(0.8 -> 'high, 0.2 -> 'medium)("power", this)
7  }
8  class V6 extends Engine {
9    val power: Element[Symbol] =
10     Select(0.2 -> 'high, 0.5 -> 'medium, 0.3 -> 'low)("power", this)
11 }
12 object Engine1 extends V8 {
13   override val power: Element[Symbol] = Constant('high)("power", this)
14 }
15 class Car extends ElementCollection {
16   val engine = Uniform[Engine](new V8, new V6, Engine1)("engine", this)
17   val speed = CPD(
18     get[Symbol]("engine.power"),
19     'high -> Constant(90.0),
20     'medium -> Constant(80.0),
21     'low -> Constant(70.0))
22 }
```

Lines 1–3 define an abstract Engine class with a power element and make it an element collection. Lines 4–14 define two concrete subclasses of engine, as well as a concrete instance Engine1. Lines 15–22 define a Car class. The first attribute of Car is an engine, whose value is an instance of the Engine class, making it an element collection. For the second attribute of Car, we create a CPD (conditional probability distribution) that gets the value of the reference engine.power and chooses a speed based on it. The element referred to be by engine.power is uncertain, but Figaro correctly returns 85.6667 for the expectation of the cars speed using variable elimination.

Our final example, below, illustrates how a physical differential equation model can be embedded inside a Figaro program. This snippet, which models an electromechanical motor, introduces three constants in lines 1–3 and three elements in lines 4–6. Note that the angular velocity has a sinusoidal distribution. Lines 8–12 implement the equation for the derivative of current using Apply.

```
1  val inductance = 2.51
2  val terminalResistance = 2.96
3  val motorConstant = 0.21
4  val voltage = Normal(0, 0.5)
5  val current = Normal(45, 4)
6  val angularVelocity =
7    Apply(Uniform(0, 2 * math.Pi), (d: Double) => math.sin(d))
8  val dCurrentDT =
9    Apply(voltage, current, angularVelocity,
10         (v: Double, c: Double, aV: Double) =>
```

```
            (1.0 / inductance) *
            (v - terminalResistance * c - motorConstant * aV))
```

4 Recent Research

We have recently added structured factored inference (SFI) as an experimental new framework for inference in Figaro. SFI is an approach to inference that uses the structure of the program to hierarchically decompose the inference task into components. In a Figaro program, a Chain construct has the form

```
val child = Chain(parent, f)
```

where `f` is a function that maps a parent value to a new random variable. Applying the function `f` to a value of the parent identifies a new sub–problem to be solved. Once the sub–problem has been solved, it can be replaced by a factor that relates the parent, the child, and any global variables appearing in the definition of `f`. For example, consider the Figaro model:

```
val f = (b: Boolean) => {
  val x1 = if (b) Flip(0.9) else Flip(0.1)
  val y1 = x1 && Flip(0.8)
  val x2 = if (b) Flip(0.9) else Flip(0.1)
  val y2 = x2 && Flip(0.8)
  y1 || y2
}
val a = Flip(0.6)
val b = Chain(a, f)
val c = Chain(a, f)
```

This program can be broken down into four sub–problems; one for each value of `a` applied to `f` (e.g., true and false) and for both `b` and `c`. SFI offers several benefits over unstructured inference framework:

- The ability to solve small sub–problems and build up a larger solution from those, taking advantage of the fact that interfaces into sub–problems can often be small. In many cases, the solution to the sub–problem is a factor over only the two variables representing the parent and child.
- The ability to reuse work when the same sub–problem appears multiple times in a larger problem. When the same function `f` is applied to the same parent value, we can detect that the same sub–problem is appearing and reuse the solution if it has already been generated. In particular, this reuse supports the automatic implementation of dynamic programming algorithms for framework like probabilistic context free grammars and dynamic models.
- The ability to optimize each of the sub–problems individually, using a different solver for each of the sub–problems. For example, we have shown that using simple heuristics, we can choose to apply variable elimination, belief propagation, or Gibbs sampling to a sub–problem, which results in significant performance and accuracy improvements.
- The ability to intelligently determine when a sub–problem should be solved in a nested compositional manner and when it should be incorporated into the higher level problem. The latter might be the case if the interface size is large, i.e., there are many global variables appearing in the sub–problem.

Acknowledgments

This work was funded by DARPA contract FA8750-14-C-0011. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement A (Approved for Public Release, Distribution Unlimited).

References

[1] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

[2] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 6, pp. 721–741, 1984.

[3] D. Koller and A. Pfeffer, "Object-oriented bayesian networks," in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1997, pp. 302–313.

[4] P. Domingos and M. Richardson, "1 markov logic: A unifying framework for statistical relational learning," *Statistical Relational Learning*, p. 339, 2007.

[5] E. Charniak, *Statistical language learning*. MIT press, 1996.

[6] D. Koller, D. McAllester, and A. Pfeffer, "Effective bayesian inference for stochastic programs," in *AAAI/IAAI*, 1997, pp. 740–747.

[7] A. Pfeffer, "IBAL: A probabilistic rational programming language," in *International Joint Conference on Artificial Intelligence*, 2001.

[8] ——, "Creating and manipulating probabilistic programs with Figaro," in *2nd International Workshop on Statistical Relational AI*, 2012.

[9] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov, "Blog: Probabilistic models with unknown objects," in *Introduction to statistical relational learning*, L. Getoor and B. Taskar, Eds. The MIT press, 2007, p. 373.

[10] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum, "Church: a language for generative models with non-parametric memoization and approximate inference," in *Uncertainty in Artificial Intelligence*, 2008.

[11] Microsoft Research, "Infer.net api documentation," http://research.microsoft.com/en-us/um/cambridge/projects/infernet/, 2013.

[12] N. Ramsey and A. Pfeffer, "Stochastic lambda calculus and monads of probability distributions," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 154–165.