Functional Probabilistic Programming
CUFP 2013

Avi Pfeffer
Charles River Analytics
apfeffer@cra.com

Outline

- What is probabilistic programming?
- History
- Our Figaro language
- Examples

The Problem

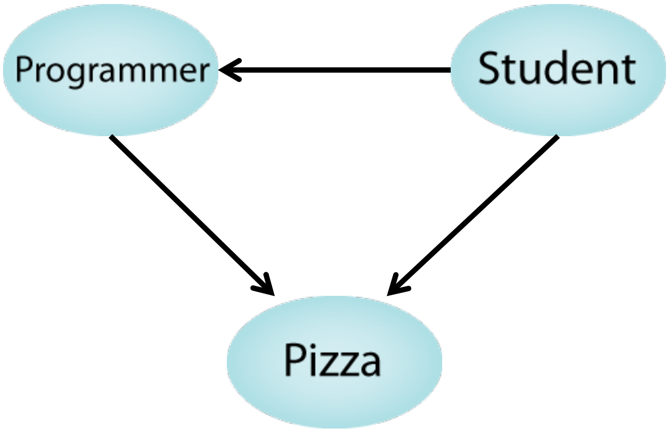
- Suppose you have some information
  - E.g., Brian ate pizza last night
- You want to answer some questions based on this information
  - Is Brian a student?
  - Is Brian a programmer?
- There is uncertainty in the answers

Probabilistic Modeling

- Create a joint probability distribution over the variables
  - P(Pizza, programmer, student)
  - Either directly or by learning it from data
- Assert the evidence
  - Brian ate pizza
- Use probabilistic inference to get the answer
  - P(student, programmer | pizza)

Generative Models

- Probabilistic models in which variables are generated in order
  - Later variables can depend on earlier variables



- Large number of variants, e.g.
  - Bayesian networks
  - Hidden Markov models
  - Probabilistic context free grammars
  - Kalman filters
  - Probabilistic relational models

Building Generative Models

Developing a new model requires implementing

- Representation
- Inference algorithm
- Learning algorithm

- All three are significant challenges
  - Considered paper worthy

Can we make this easier?

Probabilistic Programming Systems

- Expressive representation language
  - Capture wide variety of probabilistic models
- Built-in inference and learning algorithms
  - Automatically apply to models written in the language

Functional Probabilistic Programming

- Ordinary functional language: an expression describes a computation that produces a value

```
let student = true in
let programmer = student in
let pizza = student && programmer in
(student, programmer, pizza)
```

- Functional probabilistic programming language: an expression describes a *random* computation that produces a value

```
let student = flip(0.7) in
let programmer = if (student) flip(0.2) else flip(0.1) in
let pizza =
  if (student && programmer) flip(0.9) else flip(0.3) in
(student, programmer, pizza)
```

Sampling Semantics

```
let student = flip(0.7) in
let programmer = if (student) flip(0.2) else flip(0.1) in
let pizza =
   if (student && programmer) flip(0.9) else flip(0.3) in
(student, programmer, pizza)
```

- Imagine running this program many times
- Each run generates a sample outcome
- In each run, each outcome has some probability of being generated

- The program defines a probability distribution over outcomes

Power of Functional Probabilistic Programming

- Turing complete language + probabilistic primitives
  - Naturally express wide range of probabilistic models

- A number of general purpose algorithms have been developed
  - Structured variable elimination
  - Markov chain Monte Carlo
  - Importance sampling
  - Factor graph compilation

Making Probabilistic Programming Practical

- PPLs aim to “democratize” model building
  - One should not need extensive training in ML or AI to build and code a model
- This means that a PPL should (broadly) satisfy two main goals:
  - Usability
    - Intuitive to use
    - Common design patterns easily expressed
    - Integration into other/existing applications
    - Extensible language
    - Extensible reasoning
  - Power
    - Ability to represent a wide variety of models, data, etc
    - Powerful and practical inference techniques

History | KMP 97

- With Daphne Koller and David McAllester, we first formulated the idea of probabilistic programming

- Lisp + flip

- Convoluted inference algorithm
  - Later found to be buggy

History | IBAL (2000-2007)

- Representation
  - First practical probabilistic programming language
  - OCaml like syntax
  - Implemented in Ocaml

- Inference
  - Exact inference using structured variable elimination
  - Later implemented intelligent importance sampling

- Limitations
  - Hard to integrate with applications and data
  - No continuous variables

History | Figaro (2009-Present)

- Representation
  - Embedded DSL in Scala
  - Allows distributions over any data type
  - Highly expressive constraint system also allows it to express non-generative models
- Inference
  - Extensible library of inference algorithms
  - Contains many of the most popular probabilistic inference algorithms, generalized to probabilistic programs
    - E.g., variable elimination, Metropolis-Hastings, particle filtering
- New version to be released shortly
  - Parameter learning
  - Decision making
  - Improved algorithms

Goals of the Figaro Language

- Implement a PPL in a widely-used language
  - Scala is widely-used
  - Scala interoperability with Java also gives Figaro access to an even larger library
- Provide a language to describe models with interacting components
  - Object-oriented
- Provide a means to expressed directed and undirected models with general constraints
  - Functional
- Extensibility and reuse of inference algorithms
  - Object-oriented, traits

- Using Scala helps achieve all of these goals!

Basic Figaro Concepts

- **Element[T]** is class of probabilistic models over type **T**

- Atomic elements

Constant[T], Flip, Uniform, Geometric

- Compound elements built out of other elements

If(Flip(0.8), Constant(0.5), Uniform(0,1))

The Probability Monad

- **Constant[T]** is the monadic unit

- **Chain[T,U]** implements monadic bind
  - Use an **Element[T]** to generate **T**
  - Apply a function to the **T** to generate an **Element[U]**
  - Generate a **U** from the **Element[U]**

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

- **Apply[T,U]** implements monadic fmap

Apply(Uniform(0,1), (d: Double) => d * 2)

- Most Figaro compound elements implemented using monad
  - E.g., **If**

Conditions and Constraints

- Any **Element[T]** can have conditions and constraints

- Condition: function from **T** to **Boolean**
  - Specifies a property that must be satisfied for a value to have positive probability

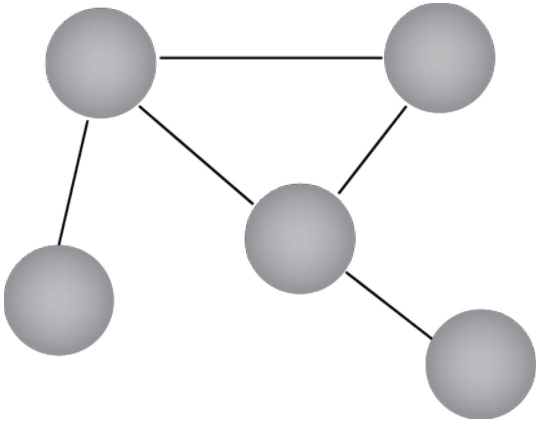
- Constraint: function from **T** to **Double**
  - Weights probability of value

- Two purposes
  - Asserting evidence
  - Specifying new kinds of models including undirected models

Example 1: Probabilistic Processes on Graphs

- Google's PageRank is a model of a probabilistic process on a graph
  - Directed edge from page A to page B if A links to B

- Consider a random walk starting at any point in the graph
  - What is the probability a node will be reached in *n* steps?



Random Walk in Figaro

- Start by defining some data structures for a webpage graph

```
class Edge(from: Int, to: Int)

class Node(ID: int, edges: Set[Edge])

class Graph(nodes: Set[Nodes]) {
  def get(id: Int) = // return Node with ID == id
}

// function that randomly builds a graph given some params
def graphGenProcess(params*): Element[Graph]
```

- Define some parameters of the random walk

```
val numSteps: Element[Int] = Constant(10)
val inputGraph: Element[Graph] = graphGenProcess(...)
val startNode: Element[Int] = Uniform(inputGraph.nodes)
```

Random Walk in Figaro

```
// randomly move forward from a node
def step(last: Int, g: Graph): Element[Int] =
  Uniform(g(last).edges.map(e => e.to))

val rWalk = Chain(inputGraph, numSteps, startNode, rFcn)

def rFcn(g: Graph, remain: Int, n: Int): Element[List[Int]] = {
 if (remain == 1)
  Apply(step(n, g), (i: Int) => List(i))
 else {
  val prev = rFcn(g, remain-1, n)
  val curr = step(Apply(prev, (l: List[Int]) => l.head), g)
  Apply(curr, prev, (i: Int, l: List[Int]) => I :: l)
 }
}
```

Example 2: Network Analysis

- People smoke with probability 0.6
- Friends are 3 times as likely to have the same smoking habit than different

- Alice is friends with Bob, Bob is friends with Clara
- Alice smokes
- What is the probability that Clara smokes?

Want a general solution that works for any friends network

Friends and Smokers | General Solution

```
// A person smokes with probability 0.6
class Person { val smokes = Flip(0.6) }

// Friends are three times as likely to have the same
// smoking habit than different
def constraint(pair: (Boolean, Boolean)) =
    if (pair._1 == pair._2) 3.0; else 1.0

// Apply the constraints to all pairs of friends
def applyConstraints(friends: List[Person]) {
  for { (p1,p2) ← friends } {
    (p1.smokes ^^ p2.smokes).addConstraint(constraint)
  }
}
```

Friends and Smokers | Specific Situation

```
// Setting up the situation
val alice, bob, clara = new Person
val friends = List((alice, bob), (bob, clara))
applyConstraints(friends)
alice.smokes.condition(true)

// Running inference and querying
val algorithm = VariableElimination(clara.smokes)
algorithm.start()
println(algorithm.probability(clara.smokes, true))
```

Example 3: Hierarchical Reasoning

- We observe an object (e.g. a vehicle on a road)
- We want to know what type of object it is
- We have some observations about it

- Inheritance hierarchies are a natural fit

Referring to Elements

- Every element
  - Has a name
  - Belongs to an element collection
    - These are implicit arguments
- A reference is a sequence of names
  - e.g., vehicle1.size
- Starting with an element collection, you can get to the element associated with a reference
  - Go through sequence of nested element collections
- There may be uncertainty in the identity of a reference
  - E.g., you don’t know what vehicle1 is
  - Figaro always resolves the reference to the *actual* element in any given world

Defining the Class Hierarchy and Properties

```
abstract class Vehicle extends ElementCollection {
  val size: Element[Symbol]
  val speed: Element[Int]
}
class Truck extends Vehicle {
  val size = Select(0.25 -> 'medium, 0.75 -> 'big)("size", this)
  val speed = Uniform(50, 60, 70)("speed", this)
}
class Pickup extends Truck {
  override val speed = Uniform(70, 80)("speed", this)
  override val size = Constant('medium)("size", this)
 }
class TwentyWheeler extends Truck ...
class Car extends Vehicle ...
```

Defining a Distribution Over Objects

```
object Vehicle {
    def generate(name: String): Element[Vehicle] =
      Dist(0.6 -> Car.generate,
              0.4 -> Truck.generate)(name, universe)
  }
object Truck {
    def generate: Element[Vehicle] =
      Dist(0.1 -> TwentyWheeler.generate,
              0.3 -> Pickup.generate,
              0.6 -> Constant[Vehicle](new Truck))
  }
object Pickup { def generate … }
object TwentyWheeler { def generate … }
object Car { def generate … }
```

Instantiating and Observing Evidence

```
val myVehicle = Vehicle.generate("v1")

universe.assertEvidence(List(NamedEvidence("v1.size",
  Observation('medium))))
```

Querying The Model

```
// Element representing the class name of the vehicle,
// e.g. Truck
val className = shortClassName(myVehicle)
val isPickup = Apply(myVehicle, (v: Vehicle) =>
  v.isInstanceOf[Pickup])

val alg = VariableElimination(isPickup, name)
alg.start()

println(alg.probability(isPickup, true))
// Print a list of class names with their probabilities
println(alg.distribution(className).toList)
```

Obtaining Figaro

- Free and open-source, available now at www.cra.com/figaro
    - Tutorial available in release

- Version 2.0 release imminent
  - Development will move to GitHub as of release
  https://github.com/p2t2

- Contact me apfeffer@cra.com or figaro@cra.com