Probabilistic Programming in Scala

Guy Van den Broeck
guy.vandenbroeck@cs.kuleuven.be

Computer Science Department
Katholieke Universiteit Leuven

September 13, 2011

What is a PPL?

- Probabilistic graphical models (Bayesian networks) important in machine learning, statistics, robotics, vision, biology, neuroscience, artificial intelligence (AI) and cognitive science.
- **P**robabilistic **P**rogramming **L**anguages unify general purpose programming with probabilistic modeling
- Examples
  - Functional, extending Scheme (Church) or Scala (Figaro, FACTORIE, ScalaPPL)
  - Logical, extending Prolog (ProbLog, PRISM, BLOG, Dyna)
  - Extending C# (Infer.NET)
- Tasks
  - Compute probabilities, most likely assignments given observations
  - Learn parameters and programs
- Applications in natural language processing, computer vision, machine learning, bioinformatics, probabilistic planning, seismic monitoring, . . .

PPLs: The Easy Case

Idea behind functional PPLs

Any function $(A, B, \ldots) => R$ can also operate on probability distributions $(Distr[A], Distr[B], \ldots) => Distr[R]$

Starting Point

Many library functions operate on Booleans:

- `&&`, `||`, `!`, `==`, `!=`
- `exists`, `forall`

Idea

1. Make a version of those functions that operates on distributions and returns a Distr[Boolean].
2. Extend with probabilistic data structures.
3. Use them to model complex probability distributions.

PPLs: The Easy Case

Idea behind functional PPLs

Any function $(A, B, \ldots) => R$ can also operate on probability distributions $(Distr[A], Distr[B], \ldots) => Distr[R]$

Starting Point

Many library functions operate on Booleans:

- `&&`, `||`, `!`, `==`, `!=`
- `exists`, `forall`

Idea

1. Make a version of those functions that operates on distributions and returns a Distr[Boolean].
2. Extend with probabilistic data structures.
3. Use them to model complex probability distributions.

PPLs: The Easy Case

Idea behind functional PPLs

Any function $(A, B, \ldots) => R$ can also operate on probability distributions $(Distr[A], Distr[B], \ldots) => Distr[R]$

Starting Point

Many library functions operate on Booleans:

- `&&`, `||`, `!`, `==`, `!=`
- `exists`, `forall`

Idea

1. Make a version of those functions that operates on distributions and returns a Distr[Boolean].
2. Extend with probabilistic data structures.
3. Use them to model complex probability distributions.

PPLs: The Easy Case

Idea behind functional PPLs

Any function $(A, B, \ldots) => R$ can also operate on probability distributions $(Distr[A], Distr[B], \ldots) => Distr[R]$

Starting Point

Many library functions operate on Booleans:

- `&&`, `||`, `!`, `==`, `!=`
- `exists`, `forall`

Idea

1. Make a version of those functions that operates on distributions and returns a Distr[Boolean].
2. Extend with probabilistic data structures.
3. Use them to model complex probability distributions.

PPLs: The Easy Case

Idea behind functional PPLs

Any function $(A, B, \ldots) => R$ can also operate on probability distributions $(Distr[A], Distr[B], \ldots) => Distr[R]$

Starting Point

Many library functions operate on Booleans:

- `&&`, `||`, `!`, `==`, `!=`
- `exists`, `forall`

Idea

1. Make a version of those functions that operates on distributions and returns a Distr[Boolean].
2. Extend with probabilistic data structures.
3. Use them to model complex probability distributions.

Examples

Boolean formulae

- Random `Variables` are objects (each object independent)

```
val a = Flip(0.3)
val b = Flip(0.6)
```

- `BooleanDistr` has member functions that build `Formula` objects.

```
val xor = a && !b || !a && b
```

- Run inference on `BooleanDistr`

```
println("Probability = " + xor.probability())
```

```
Probability = 0.54
```

Boolean formulae

- Random `Variables` are objects (each object independent)

```
val a = Flip(0.3)
val b = Flip(0.6)
```

- `BooleanDistr` has member functions that build `Formula` objects.

```
val xor = a && !b || !a && b
```

- Run inference on `BooleanDistr`

```
println("Probability = " + xor.probability())
```

```
Probability = 0.54
```

Boolean formulae

- Random `Variables` are objects (each object independent)

```
val a = Flip(0.3)
val b = Flip(0.6)
```

- `BooleanDistr` has member functions that build `Formula` objects.

```
val xor = a && !b || !a && b
```

- Run inference on `BooleanDistr`

```
println("Probability = " + xor.probability())
```

```
Probability = 0.54
```

Boolean formulae

- Random Variables are objects (each object independent)

```
val a = Flip(0.3)
val b = Flip(0.6)
```

- BooleanDistr has member functions that build Formula objects.

```
val xor = a && !b || !a && b
```

- Run inference on BooleanDistr

```
println("Probability = " + xor.probability())
```

```
Probability = 0.54
```

2-state weather HMM

```
abstract class Timestep {
  def rainy: BooleanDistr
  def umbrella = If(rainy, Flip(0.9), Flip(0.1))
}

object StartState extends Timestep {
  val rainy = Flip(0.2)
}

class SuccessorState(predecessor: Timestep) extends Timestep {
  val rainy = If(predecessor.rainy, Flip(0.5), Flip(0.1))
}
```

```
var timestep: Timestep = StartState
for(i <- 1 until 2000) timestep = new SuccessorState(timestep)
```

```
println("Probability = " + timestep.umbrella.probability())
```

2-state weather HMM

```
abstract class Timestep {
  def rainy: BooleanDistr
  def umbrella = If(rainy, Flip(0.9), Flip(0.1))
}

object StartState extends Timestep {
  val rainy = Flip(0.2)
}

class SuccessorState(predecessor: Timestep) extends Timestep {
  val rainy = If(predecessor.rainy, Flip(0.5), Flip(0.1))
}
```

```
var timestep: Timestep = StartState
for(i <- 1 until 2000) timestep = new SuccessorState(timestep)
```

```
println("Probability = " + timestep.umbrella.probability())
```

2-state weather HMM

```
abstract class Timestep {
  def rainy: BooleanDistr
  def umbrella = If(rainy, Flip(0.9), Flip(0.1))
}

object StartState extends Timestep {
  val rainy = Flip(0.2)
}

class SuccessorState(predecessor: Timestep) extends Timestep {
  val rainy = If(predecessor.rainy, Flip(0.5), Flip(0.1))
}
```

```
var timestep: Timestep = StartState
for(i <- 1 until 2000) timestep = new SuccessorState(timestep)
```

```
println("Probability = " + timestep.umbrella.probability())
```

Probabilistic Data Structures

- Probabilistic List: objects are in the list with a certain probability, given by a `BooleanDistr`

```
trait ListDistr[T] extends Distribution[List[T]]{
    def forall(f: T => BooleanDistr) : BooleanDistr
    def exists(f: T => BooleanDistr) : BooleanDistr
}
```

- Replace all `Boolean` by `BooleanDistr` in member functions
- Many possibilities (Set,Tree,...)

Probabilistic Graphs

- Viral marketing
- Learning biological pathways
- Spread of influence in social networks

```
class Person {
  val influencedFriends = new ListDistr[Person]
  def influences(target: Person): BooleanDistr = {
      if(target == this) True
      else friends.exists(_.influences(target))
  }
}
```

```
val p1,p2,p3,p4,p5,p6 = new Person

val influence1to2 = Flip(0.9)
n1.influencedFriends += (influence1to2, p2)
n2.influencedFriends += (influence1to2, p1)
...
```

```
println("Probability = " + p1.influences(p4).probability())
```

Probabilistic Graphs

- Viral marketing
- Learning biological pathways
- Spread of influence in social networks

```
class Person {
  val influencedFriends = new ListDistr[Person]
  def influences(target: Person): BooleanDistr = {
      if(target == this) True
      else friends.exists(_.influences(target))
  }
}
```

```
val p1,p2,p3,p4,p5,p6 = new Person

val influence1to2 = Flip(0.9)
n1.influencedFriends += (influence1to2, p2)
n2.influencedFriends += (influence1to2, p1)
...
```

```
println("Probability = " + p1.influences(p4).probability())
```

Probabilistic Graphs

- Viral marketing
- Learning biological pathways
- Spread of influence in social networks

```
class Person {
  val influencedFriends = new ListDistr[Person]
  def influences(target: Person): BooleanDistr = {
      if(target == this) True
      else friends.exists(_.influences(target))
  }
}
```

```
val p1,p2,p3,p4,p5,p6 = new Person

val influence1to2 = Flip(0.9)
n1.influencedFriends += (influence1to2, p2)
n2.influencedFriends += (influence1to2, p1)
...
```

```
println("Probability = " + p1.influences(p4).probability())
```

Probabilistic Graphs

- Viral marketing
- Learning biological pathways
- Spread of influence in social networks

```
class Person {
  val influencedFriends = new ListDistr[Person]
  def influences(target: Person): BooleanDistr = {
      if(target == this) True
      else friends.exists(_.influences(target))
  }
}
```

```
val p1,p2,p3,p4,p5,p6 = new Person

val influence1to2 = Flip(0.9)
n1.influencedFriends += (influence1to2, p2)
n2.influencedFriends += (influence1to2, p1)
...
```

```
println("Probability = " + p1.influences(p4).probability())
```

Probabilistic Values

- Model any discrete distribution

```
class ValDistr[T] extends Distribution[T]{
  def ==(v: T): BooleanDistr
  def map[R](f: T => ValDistr[R]): ValDistr[R]
}
```

- Apply any deterministic function to ValDistr arguments

```
def Apply[A,R](f: (A) => R)(a: ValDistr[A]): ValDistr[R]
def Apply[A,B,R](f: (A,B) => R) ...
...
```

Example

- Two dice: probability that their sum is 8?

```
val die1 = Uniform(1 to 6)
val die2 = Uniform(1 to 6)

val sum = Apply(_+_)(die1,die2)

println("Probability = " + (sum == 8).probability())
```

Probabilistic Values

- Model any discrete distribution

```
class ValDistr[T] extends Distribution[T]{
  def ==(v: T): BooleanDistr
  def map[R](f: T => ValDistr[R]): ValDistr[R]
}
```

- Apply any deterministic function to `ValDistr` arguments

```
def Apply[A,R](f: (A) => R)(a: ValDistr[A]): ValDistr[R]
def Apply[A,B,R](f: (A,B) => R) ...
...
```

Example

- Two dice: probability that their sum is 8?

```
val die1 = Uniform(1 to 6)
val die2 = Uniform(1 to 6)

val sum = Apply(_+_)(die1,die2)

println("Probability = " + (sum == 8).probability())
```

Probabilistic Values

- Model any discrete distribution

```
class ValDistr[T] extends Distribution[T]{
  def ==(v: T): BooleanDistr
  def map[R](f: T => ValDistr[R]): ValDistr[R]
}
```

- Apply any deterministic function to `ValDistr` arguments

```
def Apply[A,R](f: (A) => R)(a: ValDistr[A]): ValDistr[R]
def Apply[A,B,R](f: (A,B) => R) ...
...
```

Example

- Two dice: probability that their sum is 8?

```
val die1 = Uniform(1 to 6)
val die2 = Uniform(1 to 6)

val sum = Apply(_+_)(die1,die2)

println("Probability = " + (sum == 8).probability())
```

3-state weather HMM

```
object Sunny extends Weather
object Foggy extends Weather
object Rainy extends Weather

abstract class Timestep {
  def weather: ValDistr[Weather]
  def umbrella = weather.map{
    case Sunny => Flip(0.1)
    case Foggy => Flip(0.3)
    case Rainy => Flip(0.8)
  }
}

object StartState extends Timestep {
  val weather = ValDistr((0.3,Sunny), (0.3,Foggy), (0.4,Rainy))
}

class SuccessorState(predecessor: Timestep) extends Timestep {
  val weather = predecessor.weather.map{
    case Sunny => ValDistr((0.8,Sunny), (0.15,Foggy), (0.05,Rainy))
    case Foggy => ValDistr((0.05,Sunny), (0.3,Foggy), (0.65,Rainy))
    case Rainy => ValDistr((0.15,Sunny), (0.35,Foggy), (0.5,Rainy))
  }
}
```

Why Scala for Probabilistic Programming?

- Probabilistic programming as a library
  - No separate compiler or VM
- mixin DSL
- Higher-order functions and generics
  - Pass existing deterministic code to probabilistic model
  - Memoization for efficient inference
- Object-oriented easy to model probabilistic databases
- Other advantages carry over to the probabilistic case

Thanks