Deployable Probabilistic Programming

David Tolpin
david.tolpin@gmail.com

Probabilistic Programming

Programs as statistical models:

WebPPL

```
var breastCancer = flip(0.01)
var benignCyst = flip(0.2)
var positiveMammogram =
      (breastCancer && flip(0.8))
      || (benignCyst && flip(0.5))
condition(positiveMammogram)
return breastCancer
```

Anglican

```
(defquery linearRegression [xs ys x*]
  (let [m     (sample (normal 0 2))
        b     (sample (normal 0 2))
        sigma (sample (gamma 1 1))
        f     (fn [x] (+ (* m x) b))]
    (loop [data (map #(-> [%1 %2]) xs ys)]
     (when (seq data)
      (let [[[x y] & data] data]
        (observe (normal (f x) sigma) y)
        (recur data))))
    (predict (f x*))))
```

Probabilistic Programs

- Compute *posterior distributions.*
- May contain conditions, loops, recursions.
- Are (mostly) deterministic!
- Must be run `backwards'.
- (Approximate) inference algorithms are required.

Probabilistic Programming *Languages*

- 45 PPLs listed on Wikipedia.
- 18 PPLs participated in PROBPROG 2018.
- 8 of the latter are not on Wikipedia (yet).

To name a few:

- Anglican
- BLOG
- Brich
- Church
- Edward
- Gen
- Infer.NET
- Stan
- Turing
- WebPPL
- ...

Implementation

- Infergo — http://bitbucket.org/dtolpin/infergo
- Infergo studies — http://bitbucket.org/dtolpin/infergo-studies
- GoGP, a Gaussian process library — http://bitbucket.org/dtolpin/gogp

Performance

model	time, seconds					
	compilation			execution		
	Infergo	Turing	Stan	Infergo	Turing	Stan
Eight schools	0.50	–	50	0.60	2.8	0.12
Gaussian mixture model	0.50	–	54	32	14	4.9
Latent Dirichlet allocation	0.50	–	54	8.9	12	3.7

Challenges

- *Inferentiable* programming
- Simulation vs. inference
- Data
- Deployment

Guidelines

- One language for system and model
- Common data structures
- Inference code re-used in simulation

Infergo — http://infergo.org/

- Models are written in Go.
- Relies on automatic differentiation for inference.
- Works anywhere where Go does.
- No external dependencies.
- Licensed under the MIT license.

```
type Model struct {
    Data []float64
}

func (m *Model) Observe(x []float64) float64 {
  // Our prior is a unit normal ...
  ll := Normal.Logps(0, 1, x...)
  // ... but the posterior is based on the data.
  ll += Normal.Logps(x[0], math.Exp(x[1]),
                     m.Data...)
  return ll
}
```

Why Go?

- Comes with parser and type checker.
- Compiles and runs fast.
- Allows efficient parallel execution, via *goroutines.*
- Popular for server-side programming.

Automatic differentiation

- Reverse-mode autodiff via source code transformation.
- Automatic selective differentation of models.
- Use of bultin floating point type.

```
func (m *Model) Observe(x []float64) float64 {
  var ll float64
  ad.Assignment(&ll, ad.Call(func(_ []float64) {
    Normal.Logps(0, 0, x...)
  }, 2, ad.Value(0), ad.Value(1)))
  ad.Assignment(&ll,
    ad.Arithmetic(ad.OpAdd, &ll,
      ad.Call(func(_ []float64) {
          Normal.Logps(0, 0, m.Data...)
        }, 2, &x[0],
        ad.Elemental(math.Exp, &x[1]))))
  return ad.Return(&ll)
}
```

Model composition

```
type A struct {Data []float64}
func (model A) Observe(x []float64) {
  ...
}

type B struct {Data []float64}
func (model B) Observe(x []float64) {
  ...
}

type AB struct {Data []float64}
func (model AB) Observe(x []float64)
    float64 {
  return A{model.Data}.Observe(x[:1]) +
    B{model.Data}.Observe(x[1:])
}
```

Streaming & stochasticity

```
type StreamingModel struct {
  Data chan float64  // data is a channel
  N    int           // batch size
}

func (m *StreamingModel)
     Observe(x []float64) float64 {
  ll := Normal.Logps(0, 1, x...)
  // observe a batch of data from the channel
  for i := 0; i != m.N; i++ {
    ll += Normal.Logp(x[0], math.Exp(x[1]),
                      <- m.Data)
  }
  return ll
}
```

Case studies

8 schools

```
type Model struct {
  J          int
  Y          []float64
  Sigma      []float64
}

func (m *Model) Observe(x []float64) float64 {
  mu := x[0]
  tau := math.Exp(x[1])
  eta := x[2:]

  ll := Normal.Logp(0, 1, eta)
  for i, y := range m.Y {
    theta := mu + tau*eta[i]
    ll += Normal.Logp(theta, m.Sigma[i], y)
  }
  return ll
}
```

Linear Regression

```
type Model struct {
  Data  [][]float64
}

func (m *Model) Observe(x []float64)
    float64 {
  ll := 0.
  alpha, beta := x[0], x[1]
  sigma := math.Exp(x[2])

  for i := range m.Data {
    ll += Normal.Logp(
      m.Simulate(m.Data[i][0], alpha, beta),
      sigma, m.Data[i][1])
  }
  return ll
}

// Simulate predicts y for x based on
// inferred parameters.
func (m *Model) Simulate(x, alpha, beta float64)
    float64 {
  y := alpha +beta*x
  return y
}
```

Latent Dirichlet Allocation

```
type LDAModel struct {
  K     int         // num topics
  V     int         // num words
  M     int         // num docs
  N     int         // total word instances
  Word  []int       // word n
  Doc   []int       // doc for word n
  Alpha []float64 // topic prior
  Beta  []float64 // word prior
}

func (m *LDAModel) Observe(x []float64) float64 {
  ll := Normal.Logps(0, 1, x...)
  theta := make([][]float64, m.M)
  D.Simplices(&x, m.K, theta)
  phi := make([][]float64, m.K)
  D.Simplices(&x, m.V, phi)

  // priors
  ll += Dirichlet{m.K}.Logps(m.Alpha, theta...)
  ll += Dirichlet{m.V}.Logps(m.Beta, phi...)

  gamma := make([]float64, m.K)
  for in := 0; in != m.N; in++ {
    for ik := 0; ik != m.K; ik++ {
      gamma[ik] =
        math.Log(theta[m.Doc[in]-1][ik]) +
        math.Log(phi[ik][m.Word[in]-1])
    }
    ll += D.LogSumExp(gamma)
  }
  return ll
}
```