Sigma Documentation

Release 0.0.1

Zenna Tavares

September 23, 2015

Contents

Sigma is a probabilistic programming environment implemented in Julia. In it, you can specify probabilistic models as normal programs, and perform inference.

Sigma is built on top of Julia but not yet in the official Julia Package repository. You can still easily install it from a Julia repl with:

```
Pkg.clone("https://github.com/zenna/Sigma.jl.git")
```

Sigma is then loaded with

```
using Sigma
```

Contents:

CHAPTER 1

Getting Started

First we need to include Sigma

```
julia> using Sigma
```

Then, we create a uniform distribution `x` and draw 100 samples from it using `rand`:

```
julia> x = uniform(0,1)
RandVar{Float64}

julia> rand(x, 100)
100-element Array{Float64,1}:
  0.376264
  0.492391
 ...
```

Then we can find the probability that `x^2` is greater than 0.6:

```
julia> prob(x^2 > 0.6)
[0.225463867187499 0.225463867187499]
```

Then we can introduce an exponentially distributed variable `y`, and find the probability that `x^2` is greater than 0.6 under the condition that the sum of `x` and `y` is less than 1

```
julia> y = exponential(0.5)
julia> prob(x^2 > 0.6, x + y < 1)
[0.053548951048950494 0.06132144691466614]
```

Then, instead of computing conditional probabilities, we can sample from `x` under the same condition:

```
julia> rand(x, x + y < 1)
0.04740462764340371
```

CHAPTER 2

Building Probabilistic Models

Building probabilistic models in Sigma is simple. A probabilistic model is simply a random variable. Sigma provides a collection of functions which construct random variables. Arguably the simplest random variable is the standard uniform, which is created by `uniform`:

```
x = uniform(0,1)
 RandVar{Float64}
```

Random variables are values of the `RandVar{T}` type, which is paramterised by `T`. There are many ways to think about random variables, but for the most part you can treat them as if they were values of the type `T`. That is, you can treat a `RandVar{Float64}` as if it were a `Float64`. For example, you can apply primitive functions to them:

```
x = uniform(0,1)
y = uniform(0,1)
x + y
  RandVar{Float64}
```

Notice `x + y` is also a random variable. When you apply functions to random variables which treat them as if they were numbers (e.g. +, −, /, ...), you will get back a random variable of the appropriate type.

Of course Sigma has random variables of type other than `Float64`. To sample from a Bernoulli distribution use `flip` (named so because it is like flipping a coin):

```
x = flip(0.6)
  RandVar{Bool}
```

Similarly, boolean functions can be applied to `RandVar{Bool}`

```
x = flip(0.3)
y = flip(0.6)
z = x & y
```

Note: Short-cut operators like `&&`, `||`, `?` and `if` cannot be used with `RandVar{Bool}`. This is a tricky limitation we are trying to solve.

With these tools we can now make a more complex model:

```
a = logistic(0.5, 0.5)
x = uniform(0,1)
y = exponential(x)

z = ifelse((y > 0.4) | flip(0.3), sin(a), atan(x+y)^3)
```

CHAPTER 3

Random Arrays

Sigma.jl provides support for multivariate random variables, i.e., random arrays. Random arrays are values of the type `RandArray{T}`.

3.1 Fixed Size Random Array

The simplest (and currently only) type of random array is essentially just a normal (dense) arrays of values of type `RandVar`. A `RandArray` is created using a primitive multivariate random variable constructor. One simple example is `mvuniform` where `mv` stands for multivariate:

```
x = Sigma.mvuniform(0,1,10)
  RandArray{Float64}
```

Here `x` is a random array of 10 **independent** random variables uniformly distributed between 0 and 1.

Most of the normal array functions can be used with a `RandArray`. For instance we can inspect its size or index it with integer indices.

```
julia> size(x)
(10,)

julia> x[1]
RandVar{Float64}
```

But a `RandArray` is also a random variable and hence we can do things like sample from it:

```
julia> rand(x)
10-element Array{Float64,1}:
 0.558689
 0.791846
 0.874605
 0.212741
 0.476137
 0.246175
 0.7308
 0.625276
 0.154833
 0.619555
```

Note:

- A `RandArray` cannot be indexed by a `RandVar{Int}`.

Like normal arrays, A `RandArray` can be created with uninitialized values:

```
Sigma.RandArray(Float64, 5,5)
```

A `RandArray` can also be initialised from a normal arrays of either constants or values of type `RandVar`, so long as they are all of the same type.

```
X = RandArray([uniform(0,1), exponential(0.5))])
Y = RandArray([1.0, 3.0])
```

CHAPTER 4

Primitive Univariate Random Variable

The following is a list of the primitive univariate random variables currently supported in Sigma. In addition to categorising random variables by their output type, we distinguish between random variables which can and cannot be expressed in closed form. This is because there are currently restrictions on where random variables without a closed form solution (e.g. normal) can be used.

4.1 Uniform Distribution

The probability density function of a Continuous Uniform distribution over an interval $[a, b]$ is

$$f(x; a, b) = \frac{1}{b-a}, \quad a \leq x \leq b$$

uniform(*a::Real*, *b::Real*)
Returns uniformly distributed random variable between a and b

```
uniform(a, b)     # Uniform distribution over [a, b]
```

4.2 Logistic Distribution

The probability density function of a Logistic distribution with location μ and scale β is

$$f(x; \mu, \beta) = \frac{1}{4\beta} \text{sech}^2 \left(\frac{x-\mu}{\beta} \right)$$

logistic(*μ::Real*, *β::Real*)
Returns logistically distributed random variable with location μ and scale β

4.3 Normal Distribution

The probability density function of a Normal distribution with mean μ and variance σ is

$$f(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left(-\frac{(x-\mu)^2}{2\sigma^2} \right)$$

normal (μ::*Real*, σ::*Real*)

Returns normally distributed random variable with location μ and scale β

```
a = normal()          # standard Normal distribution with zero mean and unit variance
b = normal(mu)        # Normal distribution with mean mu and unit variance
normal(mu, sig)       # Normal distribution with mean mu and variance sig^2
normal(a,b)           # Normal distribution with normal parameters
```

CHAPTER 5

Inference Queries

Sigma supports four kinds of inference query:

- Probability Queries - probability of `X`
- Conditional probability Queries - probability of `X` given that `Y` is true
- Sampling - sample from `X`
- Conditional Sampling: sample from `X` given that `Y` is true

5.1 Probability Queries

Probability queries are done by `prob`:

prob (*X::RandVar{Bool}*)
Return a the probability that X is true. Returns an interval $I = [a, b]$, such that $a \leq P(X) \leq b$.

```
X = uniform(0,1)
Y = uniform(0,1)
prob(X + Y > 1)
```

5.2 Conditional Probability Queries

Conditional probability queries are also done with `prob`, but it expects two boolean RandVars as input:

prob (*X::RandVar{Bool}*, *Y::RandVar{Bool}*)
Return $P(X|Y)$: the conditional probability that X is true given Y is true. Returns an interval $I = [a, b]$ such that $a \leq P(X|Y) \leq b$.

```
X = uniform(0,1)
Y = uniform(0,1)
prob(Y > 0.0,X > 0.0)
```

5.3 Probability bounds

As described above, a (conditional) probability query returns a **probability bound**, i.e. an interval with a lower and upper bound, instead of a single number. Sigma guarantees that the true answer is within these bounds.

Bounds are representing as intervals from the AbstractDomains package. If you really want a single point, you can use `mid` to get the midpoint between the ends.

```
x = prob(Y > 0.0, X > 0.0)
mid(x)
```

Note:

- Probability and conditional probability queries will only work for relatively low dimensional problems (less than around 10). You can find the dimensionality of a random variable using `ndims`.

```
julia> X = uniform(0,1)
RandVar{Float64}

julia> Y = uniform(0,1)
RandVar{Float64}

julia> Z = X + Y
RandVar{Float64}

julia> ndims(Z)
2
```

5.4 Sampling

To sample from any random variable use `rand`

rand{T}(X::RandVar{T})
Sample a value of type `T` from `X`

```
X = exponential(0.5)
rand(X)
```

5.5 Conditional Sampling

Just like `prob`, to conditionally sample use `rand` with the second argument with the `RandVar{Bool}` you want to condition on:

rand{T}(X::RandVar{T}, Y::RandVar{Bool}, n::Integer)
Sample *n* values of type `T` from `X` conditioned on `Y` being true

```
X = exponential(0.5)
rand(X, X>0.5)
```

A *RandArray* can also used for the first argument to conditionally sample from:

rand{T}(X::RandArray{T}, Y::RandVar{Bool}, n::Integer)
Sample *n* Arrays of type `Array{T}` from `X` conditioned on `Y` being true

```
    Xs = mvuniform(0,1,10)
    rand(Xs, sum(X) == 0.5)
    10-element Array{Float64,1}:
     0.997244
     0.507635
     0.503137
     0.503914
```

```
    0.504609
    0.507393
    0.500201
    0.503708
    0.501251
    0.00574937
```

Often times you want to sample from a collection of random variables conditioned on some proposition. `rand` also can take a tuple of `RandVar` s and `RandArray` s as its first argument.

`rand{T}(X::Tuple, Y::RandVar{Bool}, n::Integer)`

Sample *n* tuples of `RandVar``s or ``RandArray``s conditioned on ``Y` being true

```
Xs = mvuniform(0,1,10)
Y = logistic(0.5, 0.5)
rand((Y,Xs), sum(X) == Y)
(9.941006795107837,
[0.997761,
 0.999576,
 0.99596,
 0.997781,
 0.999121,
 0.99348,
 0.99694,
 0.998275,
 0.998735,
 0.995129])
```

Note: if the number of samples `n` is omitted, it is assumed to be 1 and only the sample (not a list of samples) is returned.

5.6 Precision

Sigma solves a relaxed version of the problem you give it. You can control how severe that relaxation is using `precision`. Both `rand` and `prob` take `precision` as a keyword argument of type `Float64`. Increasing the precision will typically make the algorithms go slower, but the answer will be more precise. For example:

```
X = flip(0.5)
Y = flip(0.5)
@time prob(X & !Y; precision = 0.1)

# A bit faster but very innacurate
julia> @time prob(X & !Y; precision = 1.0)
elapsed time: 0.005621369 seconds (33612 bytes allocated)
[1.0 1.0]

# Slower but more accurate
@time prob(X & !Y; precision = 0.0001)
elapsed time: 0.00789781 seconds (72828 bytes allocated)
[0.24999999999999994 0.24999999999999994]
```

CHAPTER 6

License

The Sigma.jl package is licensed under the MIT "Expat" License:

> Copyright (c) 2015: Zenna Tavares. > > Permission is hereby granted, free of charge, to any person obtaining > a copy of this software and associated documentation files (the > "Software"), to deal in the Software without restriction, including > without limitation the rights to use, copy, modify, merge, publish, > distribute, sublicense, and/or sell copies of the Software, and to > permit persons to whom the Software is furnished to do so, subject to > the following conditions: > > The above copyright notice and this permission notice shall be > included in all copies or substantial portions of the Software. > > THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, > EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF > MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. > IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY > CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, > TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE > SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

L

N

P

U