Distributions.jl Documentation

Release 0.6.3

JuliaStats

May 26, 2017

Contents

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Starting With a Normal Distribution . . . 3
- 1.3 Using Other Distributions . . . 4
- 1.4 Estimate the Parameters . . . 4

2 Type Hierarchy **5**
- 2.1 Sampleable . . . 5
- 2.2 Distributions . . . 6

3 Univariate Distributions **9**
- 3.1 Univariate Continuous Distributions . . . 9
- 3.2 Univariate Discrete Distributions . . . 19
- 3.3 Common Interface . . . 22

4 Truncated Distributions **27**
- 4.1 Truncated Normal Distribution . . . 28

5 Multivariate Distributions **29**
- 5.1 Common Interface . . . 29
- 5.2 Multinomial Distribution . . . 30
- 5.3 Multivariate Normal Distribution . . . 31
- 5.4 Multivariate Lognormal Distribution . . . 34
- 5.5 Dirichlet Distribution . . . 35

6 Matrix-variate Distributions **37**
- 6.1 Common Interface . . . 37
- 6.2 Wishart Distribution . . . 37
- 6.3 Inverse-Wishart Distribution . . . 38

7 Mixture Models **39**
- 7.1 Type Hierarchy . . . 39
- 7.2 Construction . . . 40
- 7.3 Common Interface . . . 41
- 7.4 Estimation . . . 41

8 Distribution Fitting **43**

8.1 Maximum Likelihood Estimation . . . 43
8.2 Sufficient Statistics . . . 44
8.3 Maximum-a-Posteriori Estimation . . . 45

9 Create New Samplers and Distributions **47**
9.1 Create a Sampler . . . 47
9.2 Create a Univariate Distribution . . . 49
9.3 Create a Multivariate Distribution . . . 50
9.4 Create a Matrix-variate Distribution . . . 52

The *Distributions* package provides a large collection of probabilistic distributions and related functions. Particularly, *Distributions* implements:

- Moments (*e.g* mean, variance, skewness, and kurtosis), entropy, and other properties
- Probability density/mass functions (pdf) and their logarithm (logpdf)
- Moment generating functions and characteristic functions
- Maximum likelihood estimation
- Posterior *w.r.t.* conjugate prior, and Maximum-A-Posteriori (MAP) estimation

Contents:

CHAPTER 1

Getting Started

Installation

The Distributions package is available through the Julia package system by running `Pkg.add("Distributions")`. Throughout, we assume that you have installed the package.

Starting With a Normal Distribution

We start by drawing 100 observations from a standard-normal random variable.

The first step is to set up the environment:

```
julia> using Distributions
julia> srand(123) # Setting the seed
```

Then, we create a standard-normal distribution `d` and obtain samples using `rand`:

```
julia> d = Normal()
Normal(μ=0.0, σ=1.0)

julia> x = rand(d, 100)
100-element Array{Float64,1}:
  0.376264
 -0.405272
 ...
```

You can easily obtain the pdf, cdf, percentile, and many other functions for a distribution. For instance, the median (50th percentile) and the 95th percentile for the standard-normal distribution are given by:

```
julia> quantile(Normal(), [0.5, 0.95])
2-element Array{Float64,1}:
 0.0
 1.64485
```

The normal distribution is parameterized by its mean and standard deviation. To draw random samples from a normal distribution with mean 1 and standard deviation 2, you write:

```
julia> rand(Normal(1, 2), 100)
```

Using Other Distributions

The package contains a large number of additional distributions of three main types:

- `Univariate`
- `Multivariate`
- `Matrixvariate`

Each type splits further into `Discrete` and `Continuous`.

For instance, you can define the following distributions (among many others):

```
julia> Binomial(p) # Discrete univariate
julia> Cauchy(u, b)  # Continuous univariate
julia> Multinomial(n, p) # Discrete multivariate
julia> Wishart(nu, S) # Continuous matrix-variate
```

In addition, you can create truncated distributions from univariate distributions:

```
julia> Truncated(Normal(mu, sigma), l, u)
```

To find out which parameters are appropriate for a given distribution `D`, you can use `fieldnames(D)`:

```
julia> names(Cauchy)
2-element Array{Symbol,1}:
 :μ
 :β
```

This tells you that a Cauchy distribution is initialized with location μ and scale β.

Estimate the Parameters

It is often useful to approximate an empirical distribution with a theoretical distribution. As an example, we can use the array `x` we created above and ask which normal distribution best describes it:

```
julia> fit(Normal, x)
Normal(μ=0.036692077201688635, σ=1.1228280164716382)
```

Since `x` is a random draw from `Normal`, it's easy to check that the fitted values are sensible. Indeed, the estimates [0.04, 1.12] are close to the true values of [0.0, 1.0] that we used to generate `x`.

CHAPTER 2

Type Hierarchy

All samplers and distributions provided in this package are organized into a type hierarchy described as follows.

Sampleable

The root of this type hierarchy is `Sampleable`. The abstract type `Sampleable` subsumes any types of objects from which one can draw samples, which particularly includes *samplers* and *distributions*. Formally, `Sampleable` is defined as

```
abstract Sampleable{F<:VariateForm,S<:ValueSupport}
```

It has two type parameters that define the kind of samples that can be drawn therefrom.

- `F <:` `VariateForm` specifies the form of the variate, which can be one of the following:

Type	A single sample	Multiple samples
`Univariate`	a scalar number	A numeric array of arbitrary shape, each element being a sample
`Multivariate`	a numeric vector	A matrix, each column being a sample
`Matrixvariate`	a numeric matrix	An array of matrices, each element being a sample matrix

- `S <:` `ValueSupport` specifies the support of sample elements, which can be either of the following:

Type	Element type	Descriptions
`Discrete`	`Int`	Samples take discrete values
`Continuous`	`Float64`	Samples take continuous real values

Multiple samples are often organized into an array, depending on the variate form.

The basic functionalities that a sampleable object provides is to *retrieve information about the samples it generates* and to *draw samples*. Particularly, the following functions are provided for sampleable objects:

length(*s*)
The length of each sample.

It always returns `1` when `s` is univariate.

size(*s*)
The size (i.e. shape) of each sample.

It always returns `()` when `s` is univariate, and `(length(s),)` when `s` is multivariate.

nsamples(*s*, *A*)
The number of samples contained in `A`.

See above for how multiple samples may be organized into a single array.

eltype(*s*)
The default element type of a sample.

This is the type of elements of the samples generated by the `rand` method. However, one can provide an array of different element types to store the samples using `rand!`.

rand(*s*)
Generate one sample from `s`.

rand(*s*, *n*)
Generate `n` samples from `s`. The form of the returned object depends on the variate form of `s`:

- When `s` is univariate, it returns a vector of length `n`.
- When `s` is multivariate, it returns a matrix with `n` columns.
- When `s` is matrix-variate, it returns an array, where each element is a sample matrix.

rand!(s, A)
Generate one or multiple samples from `s` to a pre-allocated array `A`.

`A` should be in the form as specified above. The rules are summarized as below:

- When `s` is univariate, `A` can be an array of arbitrary shape. Each element of `A` will be overriden by one sample.
- When `s` is multivariate, `A` can be a vector to store one sample, or a matrix with each column for a sample.
- When `s` is matrix-variate, `A` can be a matrix to store one sample, or an array of matrices with each element for a sample matrix.

Distributions

We use *Distribution*, a subtype of *Sampleable* as defined below, to capture probabilistic distributions. In addition to being sampleable, a *distribution* typically comes with an explicit way to combine its domain, probability density functions, among many other quantities.

```
abstract Distribution{F<:VariateForm,S<:ValueSupport} <: Sampleable{F,S}
```

To simplify the use in practice, we introduce a series of type alias as follows:

```
const UnivariateDistribution{S<:ValueSupport}    = Distribution{Univariate,S}
const MultivariateDistribution{S<:ValueSupport} = Distribution{Multivariate,S}
const MatrixDistribution{S<:ValueSupport}        = Distribution{Matrixvariate,S}
const NonMatrixDistribution = Union{UnivariateDistribution, MultivariateDistribution}
```

```
const DiscreteDistribution{F<:VariateForm}   = Distribution{F,Discrete}
const ContinuousDistribution{F<:VariateForm} = Distribution{F,Continuous}

const DiscreteUnivariateDistribution     = Distribution{Univariate,    Discrete}
const ContinuousUnivariateDistribution   = Distribution{Univariate,    Continuous}
const DiscreteMultivariateDistribution   = Distribution{Multivariate,  Discrete}
const ContinuousMultivariateDistribution = Distribution{Multivariate,  Continuous}
const DiscreteMatrixDistribution         = Distribution{Matrixvariate, Discrete}
const ContinuousMatrixDistribution       = Distribution{Matrixvariate, Continuous}
```

All methods applicable to *Sampleable* also applies to *Distribution*. The API for distributions of different variate forms are different (refer to *Univariate Distributions*, *Multivariate Distributions*, and *Matrix-variate Distributions* for details).

CHAPTER 3

Univariate Distributions

Univariate Continuous Distributions

Arcsine(a, b)

The *Arcsine distribution* has probability density function

$$f(x) = \frac{1}{\pi\sqrt{(x-a)(b-x)}}, \quad x \in [a, b]$$

```
Arcsine()          # Arcsine distribution with support [0, 1]
Arcsine(b)         # Arcsine distribution with support [0, b]
Arcsine(a, b)      # Arcsine distribution with support [a, b]

params(d)          # Get the parameters, i.e. (a, b)
minimum(d)         # Get the lower bound, i.e. a
maximum(d)         # Get the upper bound, i.e. b
location(d)        # Get the left bound, i.e. a
scale(d)           # Get the span of the support, i.e. b - a
```

External links

- Arcsine distribution on Wikipedia

Beta(α, β)

The *Beta distribution* has probability density function

$$f(x; \alpha, \beta) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1}, \quad x \in [0, 1]$$

The Beta distribution is related to the *Gamma()* distribution via the property that if $X \sim \text{Gamma}(\alpha)$ and $Y \sim \text{Gamma}(\beta)$ independently, then $X/(X+Y) \sim \text{Beta}(\alpha, \beta)$.

```
Beta()         # equivalent to Beta(1.0, 1.0)
Beta(a)        # equivalent to Beta(a, a)
Beta(a, b)     # Beta distribution with shape parameters a and b
```

```
params(d)      # Get the parameters, i.e. (a, b)
```

External links

- Beta distribution on Wikipedia

BetaPrime(α, β)

The *Beta prime distribution* has probability density function

$$f(x; \alpha, \beta) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1+x)^{-(\alpha+\beta)}, \quad x > 0$$

The Beta prime distribution is related to the `Beta()` distribution via the relation ship that if $X \sim \text{Beta}(\alpha, \beta)$ then $\frac{X}{1-X} \sim \text{BetaPrime}(\alpha, \beta)$

```
BetaPrime()         # equivalent to BetaPrime(0.0, 1.0)
BetaPrime(a)        # equivalent to BetaPrime(a, a)
BetaPrime(a, b)     # Beta prime distribution with shape parameters a and b

params(d)           # Get the parameters, i.e. (a, b)
```

External links

- Beta prime distribution on Wikipedia

Cauchy(μ, σ)

The *Cauchy distribution* with location μ and scale σ has probability density function

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

```
Cauchy()          # Standard Cauchy distribution, i.e. Cauchy(0.0, 1.0)
Cauchy(u)         # Cauchy distribution with location u and unit scale, i.e.␣
↪Cauchy(u, 1.0)
Cauchy(u, b)      # Cauchy distribution with location u and scale b

params(d)         # Get the parameters, i.e. (u, b)
location(d)       # Get the location parameter, i.e. u
scale(d)          # Get the scale parameter, i.e. b
```

External links

- Cauchy distribution on Wikipedia

Chi(ν)

The *Chi distribution* ν degrees of freedom has probability density function

$$f(x; k) = \frac{1}{\Gamma(k/2)} 2^{1-k/2} x^{k-1} e^{-x^2/2}, \quad x > 0$$

It is the distribution of the square-root of a `Chisq()` variate.

```
Chi(k)        # Chi distribution with k degrees of freedom

params(d)     # Get the parameters, i.e. (k,)
dof(d)        # Get the degrees of freedom, i.e. k
```

External links

•Chi distribution on Wikipedia

Chisq(ν)

The *Chi squared distribution* (typically written χ^2) with ν degrees of freedom has the probability density function

$$f(x; k) = \frac{x^{k/2-1} e^{-x/2}}{2^{k/2}\Gamma(k/2)}, \quad x > 0.$$

If ν is an integer, then it is the distribution of the sum of squares of ν independent standard *Normal()* variates.

```
Chisq(k)      # Chi-squared distribution with k degrees of freedom

params(d)     # Get the parameters, i.e. (k,)
dof(d)        # Get the degrees of freedom, i.e. k
```

External links

•Chi-squared distribution on Wikipedia

Erlang(α, θ)

The *Erlang distribution* is a special case of a *Gamma()* distribution with integer shape parameter.

```
Erlang()        # Erlang distribution with unit shape and unit scale, i.e.␣
↪Erlang(1.0, 1.0)
Erlang(a)       # Erlang distribution with shape parameter a and unit scale, i.e.␣
↪Erlang(a, 1.0)
Erlang(a, s)    # Erlang distribution with shape parameter a and scale b
```

External links

•Erlang distribution on Wikipedia

Exponential(θ)

The *Exponential distribution* with scale parameter θ has probability density function

$$f(x; \theta) = \frac{1}{\theta} e^{-\frac{x}{\theta}}, \quad x > 0$$

```
Exponential()       # Exponential distribution with unit scale, i.e. Exponential(1.
↪0)
Exponential(b)      # Exponential distribution with scale b

params(d)           # Get the parameters, i.e. (b,)
scale(d)            # Get the scale parameter, i.e. b
rate(d)             # Get the rate parameter, i.e. 1 / b
```

External links

•Exponential distribution on Wikipedia

FDist($\nu 1, \nu 2$)

The *F distribution* has probability density function

$$f(x; \nu_1, \nu_2) = \frac{1}{xB(\nu_1/2, \nu_2/2)} \sqrt{\frac{(\nu_1 x)^{\nu_1} \cdot \nu_2^{\nu_2}}{(\nu_1 x + \nu_2)^{\nu_1+\nu_2}}}, \quad x > 0$$

It is related to the *Chisq()* distribution via the property that if $X_1 \sim \text{Chisq}(\nu_1)$ and $X_2 \sim \text{Chisq}(\nu_2)$, then \$(X_1/\nu_1) / (X_2 / \nu_2) \sim FDist(\nu_1, \nu_2)‘.

```
FDist(d1, d2)      # F-Distribution with parameters d1 and d2

params(d)          # Get the parameters, i.e. (d1, d2)
```

External links

- F distribution on Wikipedia

Frechet (α, θ)

The *Fréchet distribution* with shape α and scale θ has probability density function

$$f(x; \alpha, \theta) = \frac{\alpha}{\theta} \left(\frac{x}{\theta}\right)^{-\alpha-1} e^{-(x/\theta)^{-\alpha}}, \quad x > 0$$

```
Frechet()          # Fréchet distribution with unit shape and unit scale, i.e.␣
↪Frechet(1.0, 1.0)
Frechet(a)         # Fréchet distribution with shape a and unit scale, i.e.␣
↪Frechet(a, 1.0)
Frechet(a, b)      # Fréchet distribution with shape a and scale b

params(d)          # Get the parameters, i.e. (a, b)
shape(d)           # Get the shape parameter, i.e. a
scale(d)           # Get the scale parameter, i.e. b
```

External links

- Fréchet_distribution on Wikipedia

Gamma (α, θ)

The *Gamma distribution* with shape parameter α and scale θ has probability density function

$$f(x; \alpha, \theta) = \frac{x^{\alpha-1} e^{-x/\theta}}{\Gamma(\alpha)\theta^{\alpha}}, \quad x > 0$$

```
Gamma()            # Gamma distribution with unit shape and unit scale, i.e.␣
↪Gamma(1.0, 1.0)
Gamma(a)           # Gamma distribution with shape a and unit scale, i.e. Gamma(a,␣
↪1.0)
Gamma(a, b)        # Gamma distribution with shape a and scale b

params(d)          # Get the parameters, i.e. (a, b)
shape(d)           # Get the shape parameter, i.e. a
scale(d)           # Get the scale parameter, i.e. b
```

External links

- Gamma distribution on Wikipedia

GeneralizedExtremeValue (μ, σ, ξ)

The *Generalized extreme value distribution* with shape parameter ξ, scale σ and location μ has probability density function

$$f(x; \xi, \sigma, \mu) = \begin{cases} \frac{1}{\sigma}\left[1 + \left(\frac{x-\mu}{\sigma}\right)\xi\right]^{-1/\xi-1} \exp\left\{-\left[1 + \left(\frac{x-\mu}{\sigma}\right)\xi\right]^{-1/\xi}\right\} & \text{for } \xi \neq 0 \\ \frac{1}{\sigma}\exp\left\{-\frac{x-\mu}{\sigma}\right\}\exp\left\{-\exp\left[-\frac{x-\mu}{\sigma}\right]\right\} & \text{for } \xi = 0 \end{cases}$$

for

$$x \in \begin{cases} \left[\mu - \frac{\sigma}{\xi}, +\infty\right) & \text{for } \xi > 0 \\ (-\infty, +\infty) & \text{for } \xi = 0 \\ \left(-\infty, \mu - \frac{\sigma}{\xi}\right] & \text{for } \xi < 0 \end{cases}$$

```
GeneralizedExtremeValue(k, s, m)      # Generalized Pareto distribution with␣
↪shape k, scale s and location m.

params(d)        # Get the parameters, i.e. (k, s, m)
shape(d)         # Get the shape parameter, i.e. k (sometimes called c)
scale(d)         # Get the scale parameter, i.e. s
location(d)      # Get the location parameter, i.e. m
```

External links

- Generalized extreme value distribution on Wikipedia

GeneralizedPareto(ξ, σ, μ)

The *Generalized Pareto distribution* with shape parameter ξ, scale σ and location μ has probability density function

$$f(x;\xi,\sigma,\mu) = \begin{cases} \frac{1}{\sigma}(1+\xi\frac{x-\mu}{\sigma})^{-\frac{1}{\xi}-1} & \text{for } \xi \neq 0 \\ \frac{1}{\sigma}e^{-\frac{(x-\mu)}{\sigma}} & \text{for } \xi = 0 \end{cases}, \quad x \in \begin{cases} [\mu,\infty] & \text{for } \xi \geq 0 \\ [\mu, \mu - \sigma/\xi] & \text{for } \xi < 0 \end{cases}$$

```
GeneralizedPareto()              # Generalized Pareto distribution with unit shape␣
↪and unit scale, i.e. GeneralizedPareto(1.0, 1.0, 0.0)
GeneralizedPareto(k, s)          # Generalized Pareto distribution with shape k␣
↪and scale s, i.e. GeneralizedPareto(k, s, 0.0)
GeneralizedPareto(k, s, m)       # Generalized Pareto distribution with shape k,␣
↪scale s and location m.

params(d)        # Get the parameters, i.e. (k, s, m)
shape(d)         # Get the shape parameter, i.e. k
scale(d)         # Get the scale parameter, i.e. s
location(d)      # Get the location parameter, i.e. m
```

External links

- Generalized Pareto distribution on Wikipedia

Gumbel(μ, θ)

The *Gumbel distribution* with location μ and scale θ has probability density function

$$f(x;\mu,\theta) = \frac{1}{\theta}e^{-(z+e^z)}, \quad \text{with } z = \frac{x-\mu}{\theta}$$

```
Gumbel()             # Gumbel distribution with zero location and unit scale, i.e.␣
↪Gumbel(0.0, 1.0)
Gumbel(u)            # Gumbel distribution with location u and unit scale, i.e.␣
↪Gumbel(u, 1.0)
Gumbel(u, b)         # Gumbel distribution with location u and scale b

params(d)         # Get the parameters, i.e. (u, b)
location(d)       # Get the location parameter, i.e. u
scale(d)          # Get the scale parameter, i.e. b
```

External links

- Gumbel distribution on Wikipedia

InverseGamma(α, θ)

The *inverse gamma distribution* with shape parameter α and scale θ has probability density function

$$f(x;\alpha,\theta) = \frac{\theta^\alpha x^{-(\alpha+1)}}{\Gamma(\alpha)}e^{-\frac{\theta}{x}}, \quad x > 0$$

It is related to the *Gamma()* distribution: if $X \sim \text{Gamma}(\alpha, \beta)$, then $1/X \sim \text{InverseGamma}(\alpha, \beta^{-1})$.

```
InverseGamma()        # Inverse Gamma distribution with unit shape and unit scale,
↪ i.e. InverseGamma(1.0, 1.0)
InverseGamma(a)       # Inverse Gamma distribution with shape a and unit scale, i.
↪e. InverseGamma(a, 1.0)
InverseGamma(a, b)    # Inverse Gamma distribution with shape a and scale b

params(d)         # Get the parameters, i.e. (a, b)
shape(d)          # Get the shape parameter, i.e. a
scale(d)          # Get the scale parameter, i.e. b
```

External links

- Inverse gamma distribution on Wikipedia

InverseGaussian (μ, λ)

The *inverse Gaussian distribution* with mean μ and shape λ has probability density function

$$f(x; \mu, \lambda) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left(\frac{-\lambda(x-\mu)^2}{2\mu^2 x}\right), \quad x > 0$$

```
InverseGaussian()              # Inverse Gaussian distribution with unit mean and␣
↪unit shape, i.e. InverseGaussian(1.0, 1.0)
InverseGaussian(mu),           # Inverse Gaussian distribution with mean mu and␣
↪unit shape, i.e. InverseGaussian(u, 1.0)
InverseGaussian(mu, lambda)    # Inverse Gaussian distribution with mean mu and␣
↪shape lambda

params(d)           # Get the parameters, i.e. (mu, lambda)
mean(d)             # Get the mean parameter, i.e. mu
shape(d)            # Get the shape parameter, i.e. lambda
```

External links

- Inverse Gaussian distribution on Wikipedia

Laplace (μ, θ)

The *Laplace distribution* with location μ and scale θ has probability density function

$$f(x; \mu, \beta) = \frac{1}{2\beta} \exp\left(-\frac{|x-\mu|}{\beta}\right)$$

```
Laplace()       # Laplace distribution with zero location and unit scale, i.e.␣
↪Laplace(0.0, 1.0)
Laplace(u)      # Laplace distribution with location u and unit scale, i.e.␣
↪Laplace(u, 1.0)
Laplace(u, b)   # Laplace distribution with location u ans scale b

params(d)       # Get the parameters, i.e. (u, b)
location(d)     # Get the location parameter, i.e. u
scale(d)        # Get the scale parameter, i.e. b
```

External links

- Laplace distribution on Wikipedia

Levy(μ, σ)

The *Lévy distribution* with location μ and scale σ has probability density function

$$f(x; \mu, \sigma) = \sqrt{\frac{\sigma}{2\pi(x-\mu)^3}} \exp\left(-\frac{\sigma}{2(x-\mu)}\right), \quad x > \mu$$

```
Levy()          # Levy distribution with zero location and unit scale, i.e. Levy(0.
↪0, 1.0)
Levy(u)         # Levy distribution with location u and unit scale, i.e. Levy(u, 1.
↪0)
Levy(u, c)      # Levy distribution with location u ans scale c

params(d)       # Get the parameters, i.e. (u, c)
location(d)     # Get the location parameter, i.e. u
```

External links

- Lévy distribution on Wikipedia

LogNormal(μ, σ)

The *log normal distribution* is the distribution of the exponential of a `Normal()` variate: if $X \sim \text{Normal}(\mu, \sigma)$ then $\exp(X) \sim \text{LogNormal}(\mu, \sigma)$. The probability density function is

$$f(x; \mu, \sigma) = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\log(x)-\mu)^2}{2\sigma^2}\right), \quad x > 0$$

```
LogNormal()           # Log-normal distribution with zero log-mean and unit scale
LogNormal(mu)         # Log-normal distribution with log-mean mu and unit scale
LogNormal(mu, sig)    # Log-normal distribution with log-mean mu and scale sig

params(d)             # Get the parameters, i.e. (mu, sig)
meanlogx(d)           # Get the mean of log(X), i.e. mu
varlogx(d)            # Get the variance of log(X), i.e. sig^2
stdlogx(d)            # Get the standard deviation of log(X), i.e. sig
```

External links

- Log normal distribution on Wikipedia

Logistic(μ, θ)

The *Logistic distribution* with location μ and scale θ has probability density function

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

```
Logistic()        # Logistic distribution with zero location and unit scale, i.e.␣
↪Logistic(0.0, 1.0)
Logistic(u)       # Logistic distribution with location u and unit scale, i.e.␣
↪Logistic(u, 1.0)
Logistic(u, b)    # Logistic distribution with location u ans scale b

params(d)         # Get the parameters, i.e. (u, b)
location(d)       # Get the location parameter, i.e. u
scale(d)          # Get the scale parameter, i.e. b
```

External links

- Logistic distribution on Wikipedia

Normal(μ, σ)

The *Normal distribution* with mean μ and standard deviation σ has probability density function

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

```
Normal()            # standard Normal distribution with zero mean and unit variance
Normal(mu)          # Normal distribution with mean mu and unit variance
Normal(mu, sig)     # Normal distribution with mean mu and variance sig^2

params(d)           # Get the parameters, i.e. (mu, sig)
mean(d)             # Get the mean, i.e. mu
std(d)              # Get the standard deviation, i.e. sig
```

External links

- Normal distribution on Wikipedia

NormalInverseGaussian$(\mu, \alpha, \beta, \delta)$

The *Normal-inverse Gaussian distribution* with location μ, tail heaviness α, asymmetry parameter β and scale δ has probability density function

$$f(x; \mu, \alpha, \beta, \delta) = \frac{\alpha\delta K_1\left(\alpha\sqrt{\delta^2 + (x-\mu)^2}\right)}{\pi\sqrt{\delta^2 + (x-\mu)^2}} \, e^{\delta\gamma + \beta(x-\mu)}$$

where K_j denotes a modified Bessel function of the third kind.

External links

- Normal-inverse Gaussian distribution on Wikipedia

Pareto(α, θ)

The *Pareto distribution* with shape α and scale θ has probability density function

$$f(x; \alpha, \theta) = \frac{\alpha\theta^\alpha}{x^{\alpha+1}}, \quad x \geq \theta$$

```
Pareto()            # Pareto distribution with unit shape and unit scale, i.e.␣
↪Pareto(1.0, 1.0)
Pareto(a)           # Pareto distribution with shape a and unit scale, i.e.␣
↪Pareto(a, 1.0)
Pareto(a, b)        # Pareto distribution with shape a and scale b

params(d)        # Get the parameters, i.e. (a, b)
shape(d)         # Get the shape parameter, i.e. a
scale(d)         # Get the scale parameter, i.e. b
```

External links * Pareto distribution on Wikipedia

Rayleigh(σ)

The *Rayleigh distribution* with scale σ has probability density function

$$f(x; \sigma) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}, \quad x > 0$$

It is related to the `Normal()` distribution via the property that if $X, Y \sim \text{Normal}(0, \sigma)$, independently, then $\sqrt{X^2 + Y^2} \sim \text{Rayleigh}(\sigma)$.

```
Rayleigh()        # Rayleigh distribution with unit scale, i.e. Rayleigh(1.0)
Rayleigh(s)       # Rayleigh distribution with scale s

params(d)         # Get the parameters, i.e. (s,)
scale(d)          # Get the scale parameter, i.e. s
```

External links

- Rayleigh distribution on Wikipedia

SymTriangularDist (μ, σ)

The *Symmetric triangular distribution* with location μ and scale σ has probability density function

$$f(x; \mu, \sigma) = \frac{1}{\sigma}\left(1 - \left|\frac{x-\mu}{\sigma}\right|\right), \quad \mu - \sigma \le x \le \mu + \sigma$$

```
SymTriangularDist()          # Symmetric triangular distribution with zero␣
↪location and unit scale
SymTriangularDist(u)         # Symmetric triangular distribution with location u␣
↪and unit scale
SymTriangularDist(u, s)      # Symmetric triangular distribution with location u␣
↪and scale s

params(d)        # Get the parameters, i.e. (u, s)
location(d)      # Get the location parameter, i.e. u
scale(d)         # Get the scale parameter, i.e. s
```

TDist (ν)

The *Students T distribution* with ν degrees of freedom has probability density function

$$f(x; d) = \frac{1}{\sqrt{d}B(1/2, d/2)}\left(1 + \frac{x^2}{d}\right)^{-\frac{d+1}{2}}$$

```
TDist(d)       # t-distribution with d degrees of freedom

params(d)      # Get the parameters, i.e. (d,)
dof(d)         # Get the degrees of freedom, i.e. d
```

External links

Student's T distribution on Wikipedia

TriangularDist (a, b, c)

The *triangular distribution* with lower limit `a`, upper limit `b` and mode `c` has probability density function

$$f(x; a, b, c) = \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \le x \le c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \le b, \\ 0 & \text{for } b < x, \end{cases}$$

```
TriangularDist(a, b)         # Triangular distribution with lower limit a, upper␣
↪limit b, and mode (a+b)/2
TriangularDist(a, b, c)      # Triangular distribution with lower limit a, upper␣
↪limit b, and mode c
```

```
params(d)         # Get the parameters, i.e. (a, b, c)
minimum(d)        # Get the lower bound, i.e. a
maximum(d)        # Get the upper bound, i.e. b
mode(d)           # Get the mode, i.e. c
```

External links

- Triangular distribution on Wikipedia

Uniform(a, b)

The *continuous uniform distribution* over an interval $[a, b]$ has probability density function

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

```
Uniform()         # Uniform distribution over [0, 1]
Uniform(a, b)     # Uniform distribution over [a, b]

params(d)         # Get the parameters, i.e. (a, b)
minimum(d)        # Get the lower bound, i.e. a
maximum(d)        # Get the upper bound, i.e. b
location(d)       # Get the location parameter, i.e. a
scale(d)          # Get the scale parameter, i.e. b - a
```

External links

- Uniform distribution (continuous) on Wikipedia

VonMises(μ, κ)

The *von Mises distribution* with mean μ and concentration κ has probability density function

$$f(x; \mu, \kappa) = \frac{1}{2\pi I_0(\kappa)} \exp\left(\kappa \cos(x - \mu)\right)$$

```
VonMises()        # von Mises distribution with zero mean and unit concentration
VonMises(κ)       # von Mises distribution with zero mean and concentration κ
VonMises(μ, κ)    # von Mises distribution with mean μ and concentration κ
```

External links

- von Mises distribution on Wikipedia

Weibull(α, θ)

The *Weibull distribution* with shape α and scale θ has probability density function

$$f(x; \alpha, \theta) = \frac{\alpha}{\theta} \left(\frac{x}{\theta}\right)^{\alpha-1} e^{-(x/\theta)^\alpha}, \quad x \geq 0$$

```
Weibull()         # Weibull distribution with unit shape and unit scale, i.e.␣
↪Weibull(1.0, 1.0)
Weibull(a)        # Weibull distribution with shape a and unit scale, i.e.␣
↪Weibull(a, 1.0)
Weibull(a, b)     # Weibull distribution with shape a and scale b

params(d)         # Get the parameters, i.e. (a, b)
shape(d)          # Get the shape parameter, i.e. a
scale(d)          # Get the scale parameter, i.e. b
```

External links

- Weibull distribution on Wikipedia

Univariate Discrete Distributions

Bernoulli (p)

A *Bernoulli distribution* is parameterized by a success rate `p`, which takes value 1 with probability `p` and 0 with probability `1-p`.

$$P(X = k) = \begin{cases} 1 - p & \text{for } k = 0, \\ p & \text{for } k = 1. \end{cases}$$

```
Bernoulli()     # Bernoulli distribution with p = 0.5
Bernoulli(p)    # Bernoulli distribution with success rate p

params(d)       # Get the parameters, i.e. (p,)
succprob(d)     # Get the success rate, i.e. p
failprob(d)     # Get the failure rate, i.e. 1 - p
```

External links:

- Bernoulli distribution on Wikipedia

BetaBinomial (n, α, β)

A *Beta-binomial distribution* is the compound distribution of the `Binomial()` distribution where the probability of success `p` is distributed according to the `Beta()`. It has three parameters: `n`, the number of trials and two shape parameters α, β

$$P(X = k) = \binom{n}{k} B(k + \alpha, n - k + \beta)/B(\alpha, \beta), \quad \text{for } k = 0, 1, 2, \ldots, n.$$

```
BetaBinomial(n, a, b)       # BetaBinomial distribution with n trials and shape␣
↪parameters a, b

params(d)       # Get the parameters, i.e. (n, a, b)
ntrials(d)      # Get the number of trials, i.e. n
```

External links:

- Beta-binomial distribution on Wikipedia

Binomial (n, p)

A *Binomial distribution* characterizes the number of successes in a sequence of independent trials. It has two parameters: `n`, the number of trials, and `p`, the probability of success in an individual trial, with the distribution:

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}, \quad \text{for } k = 0, 1, 2, \ldots, n.$$

```
Binomial()      # Binomial distribution with n = 1 and p = 0.5
Binomial(n)     # Binomial distribution for n trials with success rate p = 0.5
Binomial(n, p)  # Binomial distribution for n trials with success rate p

params(d)       # Get the parameters, i.e. (n, p)
ntrials(d)      # Get the number of trials, i.e. n
succprob(d)     # Get the success rate, i.e. p
failprob(d)     # Get the failure rate, i.e. 1 - p
```

External links:

- Binomial distribution on Wikipedia

Categorical(*p*)

A *Categorical distribution* is parameterized by a probability vector p (of length K).

$$P(X = k) = p[k] \quad \text{for } k = 1, 2, \ldots, K.$$

```
Categorical(p)    # Categorical distribution with probability vector p

params(d)         # Get the parameters, i.e. (p,)
probs(d)          # Get the probability vector, i.e. p
ncategories(d)    # Get the number of categories, i.e. K
```

Here, p must be a real vector, of which all components are nonnegative and sum to one.

Note: The input vector p is directly used as a field of the constructed distribution, without being copied.

External links:

- Categorical distribution on Wikipedia

DiscreteUniform(*a*, *b*)

A *Discrete uniform distribution* is a uniform distribution over a consecutive sequence of integers between a and b, inclusive.

$$P(X = k) = 1/(b - a + 1) \quad \text{for } k = a, a + 1, \ldots, b.$$

```
DiscreteUniform(a, b)    # a uniform distribution over {a, a+1, ..., b}

params(d)        # Get the parameters, i.e. (a, b)
span(d)          # Get the span of the support, i.e. (b - a + 1)
probval(d)       # Get the probability value, i.e. 1 / (b - a + 1)
minimum(d)       # Return a
maximum(d)       # Return b
```

External links

- Discrete uniform distribution on Wikipedia

Geometric(*p*)

A *Geometric distribution* characterizes the number of failures before the first success in a sequence of independent Bernoulli trials with success rate p.

$$P(X = k) = p(1 - p)^k, \quad \text{for } k = 0, 1, 2, \ldots.$$

```
Geometric()     # Geometric distribution with success rate 0.5
Geometric(p)    # Geometric distribution with success rate p

params(d)       # Get the parameters, i.e. (p,)
succprob(d)     # Get the success rate, i.e. p
failprob(d)     # Get the failure rate, i.e. 1 - p
```

External links

- Geometric distribution on Wikipedia

Hypergeometric(*s*, *f*, *n*)

A *Hypergeometric distribution* describes the number of successes in n draws without replacement from a finite population containing s successes and f failures.

$$P(X = k) = \frac{\binom{s}{k}\binom{f}{n-k}}{\binom{s+f}{n}}, \quad \text{for } k = \max(0, n - f), \ldots, \min(n, s).$$

```
Hypergeometric(s, f, n)   # Hypergeometric distribution for a population with
                          # s successes and f failures, and a sequence of n trials.

params(d)        # Get the parameters, i.e. (s, f, n)
```

External links

- Hypergeometric distribution on Wikipedia

NegativeBinomial(r, p)

A *Negative binomial distribution* describes the number of failures before the `r`th success in a sequence of independent Bernoulli trials. It is parameterized by `r`, the number of successes, and `p`, the probability of success in an individual trial.

$$P(X = k) = \binom{k+r-1}{k} p^r (1-p)^k, \quad \text{for } k = 0, 1, 2, \ldots.$$

The distribution remains well-defined for any positive `r`, in which case

$$P(X = k) = \frac{\Gamma(k+r)}{k!\Gamma(r)} p^r (1-p)^k, \quad \text{for } k = 0, 1, 2, \ldots.$$

```
NegativeBinomial()         # Negative binomial distribution with r = 1 and p = 0.5
NegativeBinomial(r, p)     # Negative binomial distribution with r successes and␣
↪success rate p

params(d)        # Get the parameters, i.e. (r, p)
succprob(d)      # Get the success rate, i.e. p
failprob(d)      # Get the failure rate, i.e. 1 - p
```

External links:

- Negative binomial distribution on Wikipedia

Poisson(λ)

A *Poisson distribution* descibes the number of independent events occurring within a unit time interval, given the average rate of occurrence λ.

$$P(X = k) = \frac{\lambda^k}{k!} e^{-\lambda}, \quad \text{for } k = 0, 1, 2, \ldots.$$

```
Poisson()         # Poisson distribution with rate parameter 1
Poisson(lambda)       # Poisson distribution with rate parameter lambda

params(d)         # Get the parameters, i.e. (λ,)
mean(d)           # Get the mean arrival rate, i.e. λ
```

External links:

- Poisson distribution on Wikipedia

PoissonBinomial(p)

A *Poisson-binomial distribution* describes the number of successes in a sequence of independent trials, wherein each trial has a different success rate. It is parameterized by a vector `p` (of length `K`), where `K` is the total number of trials and `p[i]` corresponds to the probability of success of the `i`th trial.

$$P(X = k) = \sum_{A \in F_k} \prod_{i \in A} p[i] \prod_{j \in A^c} (1 - p[j]), \quad \text{for } k = 0, 1, 2, \ldots, K,$$

where F_k is the set of all subsets of k integers that can be selected from $\{1, 2, 3, ..., K\}$.

```
PoissonBinomial(p)    # Poisson Binomial distribution with success rate vector p

params(d)             # Get the parameters, i.e. (p,)
succprob(d)           # Get the vector of success rates, i.e. p
failprob(d)           # Get the vector of failure rates, i.e. 1-p
```

External links:

- Poisson-binomial distribution on Wikipedia

Skellam(*μ1*, *μ2*)

A *Skellam distribution* describes the difference between two independent `Poisson()` variables, respectively with rate $\mu 1$ and $\mu 2$.

$$P(X = k) = e^{-(\mu_1+\mu_2)} \left(\frac{\mu_1}{\mu_2}\right)^{k/2} I_k(2\sqrt{\mu_1\mu_2}) \quad \text{for integer } k$$

where I_k is the modified Bessel function of the first kind.

```
Skellam(mu1, mu2)    # Skellam distribution for the difference between two Poisson
↪variables,
                     # respectively with expected values mu1 and mu2.

params(d)            # Get the parameters, i.e. (mu1, mu2)
```

External links:

- Skellam distribution on Wikipedia

Univariate distributions are the distributions whose variate forms are `Univariate` (*i.e* each sample is a scalar). Abstract types for univariate distributions:

```
const UnivariateDistribution{S<:ValueSupport} = Distribution{Univariate,S}

const DiscreteUnivariateDistribution   = Distribution{Univariate, Discrete}
const ContinuousUnivariateDistribution = Distribution{Univariate, Continuous}
```

Common Interface

A series of methods are implemented for each univariate distribution, which provide useful functionalities such as moment computation, pdf evaluation, and sampling (*i.e.* random number generation).

Parameter Retrieval

params(*d*)

Return a tuple of parameters.

Note: Let `d` be a distribution of type `D`, then `D(params(d)...)` will construct exactly the same distribution as `d`.

succprob(*d*)

Get the probability of success.

failprob(*d*)

Get the probability of failure.

scale(*d*)
Get the scale parameter.

location(*d*)
Get the location parameter.

shape(*d*)
Get the shape parameter.

rate(*d*)
Get the rate parameter.

ncategories(*d*)
Get the number of categories.

ntrials(*d*)
Get the number of trials.

dof(*d*)
Get the degrees of freedom.

Note: `params` are defined for all univariate distributions, while other parameter retrieval methods are only defined for those distributions for which these parameters make sense. See below for details.

Computation of statistics

Let `d` be a distribution:

mean(*d*)
Return the expectation of distribution `d`.

var(*d*)
Return the variance of distribution `d`.

std(*d*)
Return the standard deviation of distribution `d`, i.e. `sqrt(var(d))`.

median(*d*)
Return the median value of distribution `d`.

modes(*d*)
Return an array of all modes of `d`.

mode(*d*)
Return the mode of distribution `d`. If `d` has multiple modes, it returns the first one, i.e. `modes(d)[1]`.

skewness(*d*)
Return the skewness of distribution `d`.

kurtosis(*d*)
Return the excess kurtosis of distribution `d`.

isplatykurtic(*d*)
Return whether `d` is platykurtic (*i.e* `kurtosis(d) > 0`).

isleptokurtic(*d*)
Return whether `d` is leptokurtic (*i.e* `kurtosis(d) < 0`).

ismesokurtic(*d*)
Return whether `d` is mesokurtic (*i.e* `kurtosis(d) == 0`).

entropy(*d*)
Return the entropy value of distribution `d`.

entropy(*d*, *base*)
Return the entropy value of distribution `d`, w.r.t. a given base.

mgf(*d*, *t*)
Evaluate the moment generating function of distribution `d`.

cf(*d*, *t*)
Evaluate the characteristic function of distribution `d`.

Probability Evaluation

insupport(*d*, *x*)
When `x` is a scalar, it returns whether x is within the support of `d`. When `x` is an array, it returns whether every element in x is within the support of `d`.

pdf(*d*, *x*)
The pdf value(s) evaluated at `x`.

pdf(*d*, *rgn*)
Get/compute the probabilities over a range of values. Here, `rgn` should be in the form of `a:b`.

Note: computing the probabilities over a contiguous range of values can take advantage of the recursive relations between probability masses and thus is often more efficient than computing these probabilities individually.

pdf(*d*)
Get/compute the entire probability vector of `d`. This is equivalent to `pdf(d, minimum(d):maximum(d))`.

Note: this method is only defined for *bounded* distributions.

logpdf(*d*, *x*)
The logarithm of the pdf value(s) evaluated at x, i.e. `log(pdf(x))`.

Note: The internal implementation may directly evaluate logpdf instead of first computing pdf and then taking the logarithm, for better numerical stability or efficiency.

loglikelihood(*d*, *x*)
The log-likelihood of distribution `d` w.r.t. all samples contained in array `x`.

cdf(*d*, *x*)
The cumulative distribution function evaluated at `x`.

logcdf(*d*, *x*)
The logarithm of the cumulative function value(s) evaluated at `x`, i.e. `log(cdf(x))`.

ccdf(*d*, *x*)
The complementary cumulative function evaluated at `x`, i.e. `1 - cdf(d, x)`.

logccdf(*d*, *x*)
The logarithm of the complementary cumulative function values evaluated at x, i.e. `log(ccdf(x))`.

quantile(*d*, *q*)
The quantile value. Let `x = quantile(d, q)`, then `cdf(d, x) = q`.

cquantile(*d*, *q*)
The complementary quantile value, i.e. `quantile(d, 1-q)`.

invlogcdf(*d*, *lp*)
The inverse function of logcdf.

invlogccdf(*d*, *lp*)
The inverse function of logccdf.

Vectorized evaluation

Vectorized computation and inplace vectorized computation are supported for the following functions:

- `pdf`
- `logpdf`
- `cdf`
- `logcdf`
- `ccdf`
- `logccdf`
- `quantile`
- `cquantile`
- `invlogcdf`
- `invlogccdf`

For example, when `x` is an array, then `r = pdf(d, x)` returns an array `r` of the same size, such that `r[i] = pdf(d, x[i])`. One can also use `pdf!` to write results to pre-allocated storage, as `pdf!(r, d, x)`.

Sampling (Random number generation)

rand(*d*)
Draw a sample from d

rand(*d*, *n*)
Return a vector comprised of n independent samples from the distribution `d`.

rand(*d*, *dims*)
Return an array of size `dims` that is filled with independent samples from the distribution `d`.

rand(*d*, *dim0*, *dim1*, ...)
Similar to `rand(d, dims)` above, except that the dimensions can be input as individual integers.

For example, one can write `rand(d, 2, 3)` or `rand(d, (2, 3))`, which are equivalent.

rand!(d, arr)
Fills a pre-allocated array `arr` with independent samples from the distribution `d`.

CHAPTER 4

Truncated Distributions

The package provides a type, named *Truncated*, to represented truncated distributions, which is defined as below:

```
immutable Truncated{D<:UnivariateDistribution,S<:ValueSupport} <: Distribution
↪{Univariate,S}
    untruncated::D      # the original distribution (untruncated)
    lower::Float64      # lower bound
    upper::Float64      # upper bound
    lcdf::Float64       # cdf of lower bound
    ucdf::Float64       # cdf of upper bound

    tp::Float64         # the probability of the truncated part, i.e. ucdf - lcdf
    logtp::Float64      # log(tp), i.e. log(ucdf - lcdf)
end
```

A truncated distribution can be constructed using the constructor `Truncated` as follows:

`Truncated(d, l, u)`:
Construct a truncated distribution.

Parameters

- **`d`** – The original distribution.
- **`l`** – The lower bound of the truncation, which can be a finite value or *-Inf*.
- **`u`** – The upper bound of the truncation, which can be a finite value of *Inf*.

Many functions, including those for the evaluation of pdf and sampling, are defined for all truncated univariate distributions:

- `maximum`
- `minimum`
- `insupport`
- `pdf`
- `logpdf`

- `cdf`
- `logcdf`
- `ccdf`
- `logccdf`
- `quantile`
- `cquantile`
- `invlogcdf`
- `invlogccdf`
- `rand`
- `rand!`
- `median`

However, functions to compute statistics, such as `mean`, `mode`, `var`, `std`, and `entropy`, are not available for generic truncated distributions. Generally, there are no easy ways to compute such quantities due to the complications incurred by truncation.

Truncated Normal Distribution

The *truncated normal distribution* is a particularly important one in the family of truncated distributions. We provide additional support for this type.

One can construct a truncated normal distribution using the common constructor *Truncated*, as

```
Truncated(Normal(mu, sigma), l, u)
```

or using a dedicated constructor *TruncatedNormal* as

```
TruncatedNormal(mu, sigma, l, u)
```

Also, we provide additional methods to compute various statistics for truncated normal:

- `mean`
- `mode`
- `modes`
- `var`
- `std`
- `entropy`

CHAPTER 5

Multivariate Distributions

Multivariate distributions are the distributions whose variate forms are `Multivariate` (*i.e* each sample is a vector). Abstract types for multivariate distributions:

```
const MultivariateDistribution{S<:ValueSupport} = Distribution{Multivariate,S}

const DiscreteMultivariateDistribution   = Distribution{Multivariate, Discrete}
const ContinuousMultivariateDistribution = Distribution{Multivariate, Continuous}
```

Common Interface

The methods listed as below are implemented for each multivariate distribution, which provides a consistent interface to work with multivariate distributions.

Computation of statistics

`length`(d)
: Return the sample dimension of distribution `d`.

`size`(d)
: Return the sample size of distribution `d`, *i.e* `(length(d), 1)`.

`mean`(d)
: Return the mean vector of distribution `d`.

`var`(d)
: Return the vector of component-wise variances of distribution `d`.

`cov`(d)
: Return the covariance matrix of distribution `d`.

`cor`(d)
: Return the correlation matrix of distribution `d`.

entropy(*d*)
Return the entropy of distribution d.

Probability evaluation

insupport(*d*, *x*)
If x is a vector, it returns whether x is within the support of d. If x is a matrix, it returns whether every column in x is within the support of d.

pdf(*d*, *x*)
Return the probability density of distribution d evaluated at x.

- If x is a vector, it returns the result as a scalar.
- If x is a matrix with n columns, it returns a vector r of length n, where r[i] corresponds to x[:,i] (i.e. treating each column as a sample).

pdf!(r, d, x)
Evaluate the probability densities at columns of x, and write the results to a pre-allocated array r.

logpdf(*d*, *x*)
Return the logarithm of probability density evaluated at x.

- If x is a vector, it returns the result as a scalar.
- If x is a matrix with n columns, it returns a vector r of length n, where r[i] corresponds to x[:,i].

logpdf!(r, d, x)
Evaluate the logarithm of probability densities at columns of x, and write the results to a pre-allocated array r.

loglikelihood(*d*, *x*)
The log-likelihood of distribution d w.r.t. all columns contained in matrix x.

Note: For multivariate distributions, the pdf value is usually very small or large, and therefore direct evaluating the pdf may cause numerical problems. It is generally advisable to perform probability computation in log-scale.

Sampling

rand(*d*)
Sample a vector from the distribution d.

rand(*d*, *n*)
Sample n vectors from the distribution d. This returns a matrix of size (dim(d), n), where each column is a sample.

rand!(d, x)
Draw samples and output them to a pre-allocated array x. Here, x can be either a vector of length dim(d) or a matrix with dim(d) rows.

Node: In addition to these common methods, each multivariate distribution has its own special methods, as introduced below.

Multinomial Distribution

The Multinomial distribution generalizes the *binomial distribution*. Consider n independent draws from a Categorical distribution over a finite set of size k, and let $X = (X_1, ..., X_k)$ where X_i represents the number of times the

element `i` occurs, then the distribution of `X` is a multinomial distribution. Each sample of a multinomial distribution is a k-dimensional integer vector that sums to n.

The probability mass function is given by

$$f(x; n, p) = \frac{n!}{x_1! \cdots x_k!} \prod_{i=1}^{k} p_i^{x_i}, \quad x_1 + \cdots + x_k = n$$

```
Multinomial(n, p)    # Multinomial distribution for n trials with probability vector p

Multinomial(n, k)    # Multinomial distribution for n trials with equal probabilities
                     # over 1:k
```

Multivariate Normal Distribution

The Multivariate normal distribution is a multidimensional generalization of the *normal distribution*. The probability density function of a d-dimensional multivariate normal distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ is

$$f(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

We realize that the mean vector and the covariance often have special forms in practice, which can be exploited to simplify the computation. For example, the mean vector is sometimes just a zero vector, while the covariance matrix can be a diagonal matrix or even in the form of $\sigma\mathbf{I}$. To take advantage of such special cases, we introduce a parametric type `MvNormal`, defined as below, which allows users to specify the special structure of the mean and covariance.

```
immutable MvNormal{Cov<:AbstractPDMat,Mean<:Union{Vector,ZeroVector}} <:␣
↪AbstractMvNormal
    μ::Mean
    Σ::Cov
end
```

Here, the mean vector can be an instance of either `Vector` or `ZeroVector`, where the latter is simply an empty type indicating a vector filled with zeros. The covariance can be of any subtype of `AbstractPDMat`. Particularly, one can use `PDMat` for full covariance, `PDiagMat` for diagonal covariance, and `ScalMat` for the isotropic covariance – those in the form of $\sigma\mathbf{I}$. (See the Julia package PDMats for details).

We also define a set of alias for the types using different combinations of mean vectors and covariance:

```
const IsoNormal  = MvNormal{ScalMat,  Vector{Float64}}
const DiagNormal = MvNormal{PDiagMat, Vector{Float64}}
const FullNormal = MvNormal{PDMat,    Vector{Float64}}

const ZeroMeanIsoNormal  = MvNormal{ScalMat,  ZeroVector{Float64}}
const ZeroMeanDiagNormal = MvNormal{PDiagMat, ZeroVector{Float64}}
const ZeroMeanFullNormal = MvNormal{PDMat,    ZeroVector{Float64}}
```

Construction

Generally, users don't have to worry about these internal details. We provide a common constructor `MvNormal`, which will construct a distribution of appropriate type depending on the input arguments.

`MvNormal` (*mu*, *sig*)

Construct a multivariate normal distribution with mean `mu` and covariance represented by `sig`.

Parameters

- **mu** – The mean vector, of type `Vector{T}`, with `T<:Real`.
- **sig** – The covariance, which can in of either of the following forms (with `T<:Real`):
  - an instance of a subtype of `AbstractPDMat`
  - a symmetric matrix of type `Matrix{T}`
  - a vector of type `Vector{T}`: indicating a diagonal covariance as `diagm(abs2(sig))`.
  - a real-valued number: indicating an isotropic covariance as `abs2(sig) * eye(d)`.

MvNormal(*sig*)
Construct a multivariate normal distribution with zero mean and covariance represented by `sig`.

Here, `sig` can be in either of the following forms (with `T<:Real`):

•an instance of a subtype of `AbstractPDMat`

•a symmetric matrix of type `Matrix{T}`

•a vector of type `Vector{T}`: indicating a diagonal covariance as `diagm(abs2(sig))`.

MvNormal(*d*, *sig*)
Construct a multivariate normal distribution of dimension `d`, with zero mean, and an isotropic covariance as `abs2(sig) * eye(d)`.

Note: The constructor will choose an appropriate covariance form internally, so that special structure of the covariance can be exploited.

Addition Methods

In addition to the methods listed in the common interface above, we also provide the followinig methods for all multivariate distributions under the base type `AbstractMvNormal`:

invcov(*d*)
Return the inversed covariance matrix of d.

logdetcov(*d*)
Return the log-determinant value of the covariance matrix.

sqmahal(*d*, *x*)
Return the squared Mahalanobis distance from x to the center of d, w.r.t. the covariance.

When x is a vector, it returns a scalar value. When x is a matrix, it returns a vector of length size(x,2).

sqmahal!(r, d, x)
Write the squared Mahalanbobis distances from each column of x to the center of d to r.

rand(*rng*, *d*)
rand(*rng*, *d*, *n*)
rand!(rng, d, x)
Sample from distribution `d` using the random number generator `rng`.

Canonical form

Multivariate normal distribution is an exponential family distribution, with two *canonical parameters*: the *potential vector* $\mathbf{h}$ and the *precision matrix* $\mathbf{J}$. The relation between these parameters and the conventional representation (*i.e.*

the one using mean mu and covariance $\mathbf{\Sigma}$) is:

$$\mathbf{h} = \mathbf{\Sigma}^{-1}\boldsymbol{\mu}, \quad \text{and} \quad \mathbf{J} = \mathbf{\Sigma}^{-1}$$

The canonical parameterization is widely used in Bayesian analysis. We provide a type `MvNormalCanon`, which is also a subtype of `AbstractMvNormal` to represent a multivariate normal distribution using canonical parameters. Particularly, `MvNormalCanon` is defined as:

```
immutable MvNormalCanon{P<:AbstractPDMat,V<:Union{Vector,ZeroVector}} <:␣
↪AbstractMvNormal
    μ::V    # the mean vector
    h::V    # potential vector, i.e. inv(Σ) * μ
    J::P    # precision matrix, i.e. inv(Σ)
end
```

We also define aliases for common specializations of this parametric type:

```
const FullNormalCanon = MvNormalCanon{PDMat,     Vector{Float64}}
const DiagNormalCanon = MvNormalCanon{PDiagMat, Vector{Float64}}
const IsoNormalCanon  = MvNormalCanon{ScalMat,  Vector{Float64}}

const ZeroMeanFullNormalCanon = MvNormalCanon{PDMat,     ZeroVector{Float64}}
const ZeroMeanDiagNormalCanon = MvNormalCanon{PDiagMat, ZeroVector{Float64}}
const ZeroMeanIsoNormalCanon  = MvNormalCanon{ScalMat,  ZeroVector{Float64}}
```

A multivariate distribution with canonical parameterization can be constructed using a common constructor `MvNormalCanon` as:

MvNormalCanon (h, J)
Construct a multivariate normal distribution with potential vector `h` and precision matrix represented by `J`.

Parameters

- **h** – the potential vector, of type `Vector{T}` with `T<:Real`.
- **J** – the representation of the precision matrix, which can be in either of the following forms (`T<:Real`):
  - an instance of a subtype of `AbstractPDMat`
  - a square matrix of type `Matrix{T}`
  - a vector of type `Vector{T}`: indicating a diagonal precision matrix as `diagm(J)`.
  - a real number: indicating an isotropic precision matrix as `J * eye(d)`.

MvNormalCanon (J)
Construct a multivariate normal distribution with zero mean (thus zero potential vector) and precision matrix represented by `J`.

Here, `J` represents the precision matrix, which can be in either of the following forms (`T<:Real`):

•an instance of a subtype of `AbstractPDMat`

•a square matrix of type `Matrix{T}`

•a vector of type `Vector{T}`: indicating a diagonal precision matrix as `diagm(J)`.

MvNormalCanon (d, v)
Construct a multivariate normal distribution of dimension `d`, with zero mean and a precision matrix as `v * eye(d)`.

Note: `MvNormalCanon` share the same set of methods as `MvNormal`.

Multivariate Lognormal Distribution

The Multivariate lognormal distribution is a multidimensional generalization of the *lognormal distribution*.

If $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ has a multivariate normal distribution then $\boldsymbol{Y} = \exp(\boldsymbol{X})$ has a multivariate lognormal distribution.

Mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ of the underlying normal distribution are known as the *location* and *scale* parameters of the corresponding lognormal distribution.

The package provides an implementation, `MvLogNormal`, which wraps around `MvNormal`:

```
immutable MvLogNormal <: AbstractMvLogNormal
  normal::MvNormal
end
```

Construction

`MvLogNormal` provides the same constructors as `MvNormal`. See above for details.

Additional Methods

In addition to the methods listed in the common interface above, we also provide the following methods:

`location`(d)
Return the location vector of the distribution (the mean of the underlying normal distribution).

`scale`(d)
Return the scale matrix of the distribution (the covariance matrix of the underlying normal distribution).

`median`(d)
Return the median vector of the lognormal distribution. which is strictly smaller than the mean.

`mode`(d)
Return the mode vector of the lognormal distribution, which is strictly smaller than the mean and median.

Conversion Methods

It can be necessary to calculate the parameters of the lognormal (location vector and scale matrix) from a given covariance and mean, median or mode. To that end, the following functions are provided.

`location{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix)`
Calculate the location vector (the mean of the underlying normal distribution).

If `s == :meancov`, then m is taken as the mean, and S the covariance matrix of a lognormal distribution.

If `s == :mean | :median | :mode`, then m is taken as the mean, median or mode of the lognormal respectively, and S is interpreted as the scale matrix (the covariance of the underlying normal distribution).

It is not possible to analytically calculate the location vector from e.g., median + covariance, or from mode + covariance.

`location!{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix,`
Calculate the location vector (as above) and store the result in μ

`scale{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix)`
Calculate the scale parameter, as defined for the location parameter above.

scale!{D<:AbstractMvLogNormal}(::Type{D},s::Symbol,m::AbstractVector,S::AbstractMatrix,Σ::
Calculate the scale parameter, as defined for the location parameter above and store the result in Σ.

params{D<:AbstractMvLogNormal}(::Type{D},m::AbstractVector,S::AbstractMatrix)
Return (scale,location) for a given mean and covariance

params!{D<:AbstractMvLogNormal}(::Type{D},m::AbstractVector,S::AbstractMatrix,μ::AbstractVe
Calculate (scale,location) for a given mean and covariance, and store the results in μ and Σ

Dirichlet Distribution

The Dirichlet distribution is often used the conjugate prior for Categorical or Multinomial distributions. The probability density function of a Dirichlet distribution with parameter $\alpha = (\alpha_1, \ldots, \alpha_k)$ is

$$f(x;\alpha) = \frac{1}{B(\alpha)} \prod_{i=1}^{k} x_i^{\alpha_i - 1}, \quad \text{with } B(\alpha) = \frac{\prod_{i=1}^{k} \Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{k} \alpha_i\right)}, \quad x_1 + \cdots + x_k = 1$$

```
# Let alpha be a vector
Dirichlet(alpha)          # Dirichlet distribution with parameter vector alpha

# Let a be a positive scalar
Dirichlet(k, a)           # Dirichlet distribution with parameter a * ones(k)
```

CHAPTER 6

Matrix-variate Distributions

Matrix-variate distributions are the distributions whose variate forms are `Matrixvariate` (*i.e* each sample is a matrix). Abstract types for matrix-variate distributions:

Common Interface

Both distributions implement the same set of methods:

`size`(d)
: The size of each sample from the distribution `d`.

`length`(d)
: The length (*i.e* number of elements) of each sample from the distribution `d`.

`mean`(d)
: Return the mean matrix of `d`.

`pdf`(d, x)
: Compute the probability density at the input matrix `x`.

`logpdf`(d, x)
: Compute the logarithm of the probability density at the input matrix `x`.

`rand`(d)
: Draw a sample matrix from the distribution `d`.

Wishart Distribution

The Wishart distribution is a multidimensional generalization of the Chi-square distribution, which is characterized by a degree of freedom ν, and a base matrix S.

```
Wishart(nu, S)    # Wishart distribution with nu degrees of freedom and base matrix S.
```

Inverse-Wishart Distribution

The Inverse Wishart distribution is usually used a the conjugate prior for the covariance matrix of a multivariate normal distribution, which is characterized by a degree of freedom ν, and a base matrix Φ.

```
InverseWishart(nu, P)     # Inverse-Wishart distribution with nu degrees of freedom␣
↪and base matrix P.
```

CHAPTER 7

Mixture Models

A mixture model is a probabilistic distribution that combines a set of *component* to represent the overall distribution. Generally, the probability density/mass function is given by a convex combination of the pdf/pmf of individual components, as

$$f_{mix}(x; \Theta, \pi) = \sum_{k=1}^{K} \pi_k f(x; \theta_k)$$

A *mixture model* is characterized by a set of component parameters $\Theta = \{\theta_1, \ldots, \theta_K\}$ and a prior distribution π over these components.

Type Hierarchy

This package introduces a type `MixtureModel`, defined as follows, to represent a *mixture model*:

```
abstract AbstractMixtureModel{VF<:VariateForm,VS<:ValueSupport} <: Distribution{VF,␣
↪VS}

immutable MixtureModel{VF<:VariateForm,VS<:ValueSupport,Component<:Distribution} <:␣
↪AbstractMixtureModel{VF,VS}
    components::Vector{Component}
    prior::Categorical
end

const UnivariateMixture    = AbstractMixtureModel{Univariate}
const MultivariateMixture  = AbstractMixtureModel{Multivariate}
```

Remarks:

- We introduce `AbstractMixtureModel` as a base type, which allows one to define a mixture model with different internal implementation, while still being able to leverage the common methods defined for `AbstractMixtureModel`.
- The `MixtureModel` is a parametric type, with three type parameters:

  - `VF`: the variate form, which can be `Univariate`, `Multivariate`, or `Matrixvariate`.
  - `VS`: the value support, which can be `Continuous` or `Discrete`.
  - `Component`: the type of component distributions, *e.g.* `Normal`.
- We define two aliases: `UnivariateMixture` and `MultivariateMixture`.

With such a type system, the type for a mixture of univariate normal distributions can be written as

```
MixtureModel{Univariate,Continuous,Normal}
```

Construction

A mixture model can be constructed using the constructor `MixtureModel`. Particularly, we provide various methods to simplify the construction.

`MixtureModel`(*components*, *prior*)
Construct a mixture model with a vector of components and a prior probability vector.

`MixtureModel`(*components*)
Construct a mixture model with a vector of components. All components share the same prior probability.

`MixtureModel`(*C*, *params*, *prior*)
Construct a mixture model with component type `C`, a vector of parameters for constructing the components given by `params`, and a prior probability vector.

`MixtureModel`(*C*, *params*)
Construct a mixture model with component type `C` and a vector of parameters for constructing the components given by `params`. All components share the same prior probability.

Examples

```
# constructs a mixture of three normal distributions,
# with prior probabilities [0.2, 0.5, 0.3]
MixtureModel(Normal[
   Normal(-2.0, 1.2),
   Normal(0.0, 1.0),
   Normal(3.0, 2.5)], [0.2, 0.5, 0.3])

# if the components share the same prior, the prior vector can be omitted
MixtureModel(Normal[
   Normal(-2.0, 1.2),
   Normal(0.0, 1.0),
   Normal(3.0, 2.5)])

# Since all components have the same type, we can use a simplified syntax
MixtureModel(Normal, [(-2.0, 1.2), (0.0, 1.0), (3.0, 2.5)], [0.2, 0.5, 0.3])

# Again, one can omit the prior vector when all components share the same prior
MixtureModel(Normal, [(-2.0, 1.2), (0.0, 1.0), (3.0, 2.5)])

# The following example shows how one can make a Gaussian mixture
# where all components share the same unit variance
MixtureModel(map(u -> Normal(u, 1.0), [-2.0, 0.0, 3.0]))
```

Common Interface

All subtypes of `AbstractMixtureModel` (obviously including `MixtureModel`) provide the following two methods:

components(*d*)
Get a list of components of the mixture model `d`.

probs(*d*)
Get the vector of prior probabilities of all components of `d`.

component_type(*d*)
The type of the components of `d`.

In addition, for all subtypes of `UnivariateMixture` and `MultivariateMixture`, the following generic methods are provided:

mean(*d*)
Compute the overall mean (expectation).

var(*d*)
Compute the overall variance (only for `UnivariateMixture`).

length(*d*)
The length of each sample (only for `Multivariate`).

pdf(*d*, *x*)
Evaluate the (mixed) probability density function over `x`. Here, `x` can be a single sample or an array of multiple samples.

logpdf(*d*, *x*)
Evaluate the logarithm of the (mixed) probability density function over `x`. Here, `x` can be a single sample or an array of multiple samples.

rand(*d*)
Draw a sample from the mixture model `d`.

rand(*d*, *n*)
Draw `n` samples from `d`.

rand!(d, r)
Draw multiple samples from `d` and write them to `r`.

Estimation

There are a number of methods for estimating of mixture models from data, and this problem remains an open research topic. This package does not provide facilities for estimaing mixture models. One can resort to other packages, *e.g.* [*GaussianMixtures.jl*](https://github.com/davidavdav/GaussianMixtures.jl), for this purpose.

CHAPTER 8

Distribution Fitting

This package provides methods to fit a distribution to a given set of samples. Generally, one may write

```
d = fit(D, x)
```

This statement fits a distribution of type `D` to a given dataset `x`, where `x` should be an array comprised of all samples. The fit function will choose a reasonable way to fit the distribution, which, in most cases, is maximum likelihood estimation.

Maximum Likelihood Estimation

The function `fit_mle` is for maximum likelihood estimation.

Synopsis

- **fit_mle** (D, x)

  Fit a distribution of type `D` to a given data set `x`.

  - For univariate distribution, x can be an array of arbitrary size.
  - For multivariate distribution, x should be a matrix, where each column is a sample.

- **fit_mle** (D, x, w)

  Fit a distribution of type `D` to a weighted data set `x`, with weights given by `w`.

  Here, `w` should be an array with length `n`, where `n` is the number of samples contained in `x`.

Applicable distributions

The `fit_mle` method has been implemented for the following distributions:

Univariate:

- bernoulli
- beta
- binomial
- categorical
- discreteuniform
- exponential
- normal
- gamma
- geometric
- laplace
- pareto
- poisson
- uniform

Multivariate:

- *Multinomial Distribution*
- *Multivariate Normal Distribution*
- *Dirichlet Distribution*

For most of these distributions, the usage is as described above. For a few special distributions that require additional information for estimation, we have to use modified interface:

```
fit_mle(Binomial, n, x)         # n is the number of trials in each experiment
fit_mle(Binomial, n, x, w)

fit_mle(Categorical, k, x)      # k is the space size (i.e. the number of distinct␣
↪values)
fit_mle(Categorical, k, x, w)

fit_mle(Categorical, x)         # equivalent to fit_mle(Categorical, max(x), x)
fit_mle(Categorical, x, w)
```

Sufficient Statistics

For many distributions, estimation can be based on (sum of) sufficient statistics computed from a dataset. To simplify implementation, for such distributions, we implement `suffstats` method instead of `fit_mle` directly:

```
ss = suffstats(D, x)         # ss captures the sufficient statistics of x
ss = suffstats(D, x, w)      # ss captures the sufficient statistics of a weighted␣
↪dataset

d = fit_mle(D, ss)           # maximum likelihood estimation based on sufficient stats
```

When `fit_mle` on `D` is invoked, a fallback `fit_mle` method will first call `suffstats` to compute the sufficient statistics, and then a `fit_mle` method on sufficient statistics to get the result. For some distributions, this way is not the most efficient, and we specialize the `fit_mle` method to implement more efficient estimation algorithms.

Maximum-a-Posteriori Estimation

Maximum-a-Posteriori (MAP) estimation is also supported by this package, which is implemented as part of the conjugate exponential family framework (see Conjugate Prior and Posterior).

CHAPTER 9

Create New Samplers and Distributions

Whereas this package already provides a large collection of common distributions out of box, there are still occasions where you want to create new distributions (*e.g* your application requires a special kind of distributions, or you want to contribute to this package).

Generally, you don't have to implement every API method listed in the documentation. This package provides a series of generic functions that turn a small number of internal methods into user-end API methods. What you need to do is to implement this small set of internal methods for your distributions.

Note: the methods need to be implemented are different for distributions of different variate forms.

Create a Sampler

Unlike a full fledged distributions, a sampler, in general, only provides limited functionalities, mainly to support sampling.

Univariate Sampler

To implement a univariate sampler, one can define a sub type (say `Spl`) of `Sampleable{Univariate,S}` (where `S` can be `Disrete` or `Continuous`), and provide a `rand` method, as

```
function rand(s::Spl)
    # ... generate a single sample from s
end
```

The package already implements a vectorized version of `rand!` and `rand` that repeatedly calls the he scalar version to generate multiple samples.

Multivariate Sampler

To implement a multivariate sampler, one can define a sub type of `Sampleable{Multivariate,S}`, and provide both `length` and `_rand!` methods, as

```
Base.length(s::Spl) = ... # return the length of each sample

function _rand!{T<:Real}(s::Spl, x::AbstractVector{T})
    # ... generate a single vector sample to x
end
```

This function can assume that the dimension of `x` is correct, and doesn't need to perform dimension checking.

The package implements both `rand` and `rand!` as follows (which you don't need to implement in general):

```
function _rand!(s::Sampleable{Multivariate}, A::DenseMatrix)
    for i = 1:size(A,2)
        _rand!(s, view(A,:,i))
    end
    return A
end

function rand!(s::Sampleable{Multivariate}, A::AbstractVector)
    length(A) == length(s) ||
        throw(DimensionMismatch("Output size inconsistent with sample length."))
    _rand!(s, A)
end

function rand!(s::Sampleable{Multivariate}, A::DenseMatrix)
    size(A,1) == length(s) ||
        throw(DimensionMismatch("Output size inconsistent with sample length."))
    _rand!(s, A)
end

rand{S<:ValueSupport}(s::Sampleable{Multivariate,S}) =
    _rand!(s, Vector{eltype(S)}(length(s)))

rand{S<:ValueSupport}(s::Sampleable{Multivariate,S}, n::Int) =
    _rand!(s, Matrix{eltype(S)}(length(s), n))
```

If there is a more efficient method to generate multiple vector samples in batch, one should provide the following method

```
function _rand!{T<:Real}(s::Spl, A::DenseMatrix{T})
    ... generate multiple vector samples in batch
end
```

Remember that each *column* of A is a sample.

Matrix-variate Sampler

To implement a multivariate sampler, one can define a sub type of `Sampleable{Multivariate,S}`, and provide both `size` and `_rand!` method, as

```
Base.size(s::Spl) = ... # the size of each matrix sample

function _rand!{T<:Real}(s::Spl, x::DenseMatrix{T})
```

```
    # ... generate a single matrix sample to x
end
```

Note that you can assume `x` has correct dimensions in `_rand!` and don't have to perform dimension checking, the generic `rand` and `rand!` will do dimension checking and array allocation for you.

Create a Univariate Distribution

A univariate distribution type should be defined as a subtype of `DiscreteUnivarateDistribution` or `ContinuousUnivariateDistribution`.

Following methods need to be implemented for each univariate distribution type (say `D`):

`rand`(*d::D*)
Generate a scalar sample from `d`.

`sampler`(*d::D*)
It is often the case that a sampler relies on some quantities that may be pre-computed in advance (that are not parameters themselves).

If such a more efficient sampler exists, one should provide this `sampler` method, which would be used for batch sampling.

The general fallback is `sampler(d::Distribution) = d`.

`pdf`(*d::D*, *x::Real*)
Evaluate the probability density (mass) at `x`.

Note: The package implements the following generic methods to evaluate pdf values in batch.

- `pdf!(dst::AbstractArray, d::D, x::AbstractArray)`
- `pdf(d::D, x::AbstractArray)`

If there exists more efficient routine to evaluate pdf in batch (faster than repeatedly calling the scalar version of `pdf`), then one can also provide a specialized method of `pdf!`. The vectorized version of `pdf` simply delegats to `pdf!`.

`logpdf`(*d::D*, *x::Real*)
Evaluate the logarithm of probability density (mass) at `x`.

Whereas there is a fallback implemented `logpdf(d, x) = log(pdf(d, x))`. Relying on this fallback is not recommended in general, as it is prone to overflow or underflow.

Again, the package provides vectorized version of `logpdf!` and `logpdf`. One may override `logpdf!` to provide more efficient vectorized evaluation.

Furthermore, the generic `loglikelihood` function delegates to `_loglikelihood`, which repeatedly calls `logpdf`. If there is a better way to compute log-likelihood, one should override `_loglikelihood`.

`cdf`(*d::D*, *x::Real*)
Evaluate the cumulative probability at `x`.

The package provides generic functions to compute `ccdf`, `logcdf`, and `logccdf` in both scalar and vectorized forms. One may override these generic fallbacks if the specialized versions provide better numeric stability or higher efficiency.

`quantile`(*d::D*, *q::Real*)
Evaluate the inverse cumulative distribution function at `q`.

The package provides generic functions to compute `cquantile`, `invlogcdf`, and `invlogccdf` in both scalar and vectorized forms. One may override these generic fallbacks if the specialized versions provide better numeric stability or higher efficiency.

Also a generic `median` is provided, as `median(d) = quantile(d, 0.5)`. However, one should implement a specialized version of `median` if it can be computed faster than `quantile`.

`minimum`(*d::D*)
Return the minimum of the support of `d`.

`maximum`(*d::D*)
Return the maximum of the support of `d`.

`insupport`(*d::D*, *x::Real*)
Return whether `x` is within the support of `d`.

Note a generic fallback as `insupport(d, x) = minimum(d) <= x <= maximum(d)` is provided. However, it is often the case that `insupport` can be done more efficiently, and a specialized `insupport` is thus desirable. You should also override this function if the support is composed of multiple disjoint intervals.

Vectorized versions of `insupport!` and `insupport` are provided as generic fallbacks.

It is also recommended that one also implements the following statistics functions:

- `mean`: compute the expectation.
- `var`: compute the variance. (A generic `std` is provided as `std(d) = sqrt(var(d))`).
- `modes`: get all modes (if this makes sense).
- `mode`: returns the first mode.
- `skewness`: compute the skewness.
- `kurtosis`: compute the excessive kurtosis.
- `entropy`: compute the entropy.
- `mgf`: compute the moment generating functions.
- `cf`: compute the characteristic function.

You may refer to the source file `src/univariates.jl` to see details about how generic fallback functions for univariates are implemented.

Create a Multivariate Distribution

A multivariate distribution type should be defined as a subtype of `DiscreteMultivarateDistribution` or `ContinuousMultivariateDistribution`.

Following methods need to be implemented for each univariate distribution type (say `D`):

`length`(*d::D*)
Return the length of each sample (*i.e* the dimension of the sample space).

`_rand!{T<:Real}(d::D, x::AbstractVector{T})`
Generate a vector sample to `x`.

This function does not need to perform dimension checking.

`sampler`(*d::D*)
Return a sampler for efficient batch/repeated sampling.

_logpdf{T<:Real}(d::D, x::AbstractVector{T})

Evaluate logarithm of pdf value for a given vector `x`. This function need not perform dimension checking.

Generally, one does not need to implement `pdf` (or `_pdf`). The package provides fallback methods as follows:

```
_pdf(d::MultivariateDistribution, X::AbstractVector) = exp(_logpdf(d, X))

function logpdf(d::MultivariateDistribution, X::AbstractVector)
    length(X) == length(d) ||
        throw(DimensionMismatch("Inconsistent array dimensions."))
    _logpdf(d, X)
end

function pdf(d::MultivariateDistribution, X::AbstractVector)
    length(X) == length(d) ||
        throw(DimensionMismatch("Inconsistent array dimensions."))
    _pdf(d, X)
end
```

If there are better ways that can directly evaluate pdf values, one should override `_pdf` (*NOT* `pdf`).

The package also provides generic implementation of batch evaluation:

```
function _logpdf!(r::AbstractArray, d::MultivariateDistribution, X::DenseMatrix)
    for i in 1 : size(X,2)
        @inbounds r[i] = logpdf(d, view(X,:,i))
    end
    return r
end

function _pdf!(r::AbstractArray, d::MultivariateDistribution, X::DenseMatrix)
    for i in 1 : size(X,2)
        @inbounds r[i] = pdf(d, view(X,:,i))
    end
    return r
end

function logpdf!(r::AbstractArray, d::MultivariateDistribution, X::DenseMatrix)
    size(X) == (length(d), length(r)) ||
        throw(DimensionMismatch("Inconsistent array dimensions."))
    _logpdf!(r, d, X)
end

function pdf!(r::AbstractArray, d::MultivariateDistribution, X::DenseMatrix)
    size(X) == (length(d), length(r)) ||
        throw(DimensionMismatch("Inconsistent array dimensions."))
    _pdf!(r, d, X)
end

function logpdf(d::MultivariateDistribution, X::DenseMatrix)
    size(X, 1) == length(d) ||
        throw(DimensionMismatch("Inconsistent array dimensions."))
    _logpdf!(Vector{Float64}(size(X,2)), d, X)
end

function pdf(d::MultivariateDistribution, X::DenseMatrix)
    size(X, 1) == length(d) ||
        throw(DimensionMismatch("Inconsistent array dimensions."))
    _pdf!(Vector{Float64}(size(X,2)), d, X)
```

```
end
```

Note that if there exists faster methods for batch evaluation, one should override `_logpdf!` and `_pdf!`.

Furthermore, the generic `loglikelihood` function delegates to `_loglikelihood`, which repeatedly calls `_logpdf`. If there is a better way to compute log-likelihood, one should override `_loglikelihood`.

It is also recommended that one also implements the following statistics functions:

- `mean`: compute the mean vector.
- `var`: compute the vector of element-wise variance.
- `entropy`: compute the entropy.
- `cov`: compute the covariance matrix. (`cor` is provided based on `cov`).

Create a Matrix-variate Distribution

A multivariate distribution type should be defined as a subtype of `DiscreteMatrixDistribution` or `ContinuousMatrixDistribution`.

Following methods need to be implemented for each univariate distribution type (say `D`):

`size`(*d::D*)
Return the size of each sample.

`rand{T<:Real}(d::D)`
Generate a matrix sample.

`sampler`(*d::D*)
Return a sampler for efficient batch/repeated sampling.

`_logpdf{T<:Real}(d::D, x::AbstractMatrix{T})`
Evaluate logarithm of pdf value for a given sample `x`. This function need not perform dimension checking.

Index

A

Arcsine() (built-in function), 9

B

Bernoulli() (built-in function), 19
Beta() (built-in function), 9
BetaBinomial() (built-in function), 19
BetaPrime() (built-in function), 10
Binomial() (built-in function), 19

C

Categorical() (built-in function), 20
Cauchy() (built-in function), 10
ccdf() (built-in function), 24
cdf() (built-in function), 24, 49
cf() (built-in function), 24
Chi() (built-in function), 10
Chisq() (built-in function), 11
component_type() (built-in function), 41
components() (built-in function), 41
cor() (built-in function), 29
cov() (built-in function), 29
cquantile() (built-in function), 24

D

DiscreteUniform() (built-in function), 20
dof() (built-in function), 23

E

eltype() (built-in function), 6
entropy() (built-in function), 23, 24, 29
Erlang() (built-in function), 11
Exponential() (built-in function), 11

F

failprob() (built-in function), 22
FDist() (built-in function), 11
Frechet() (built-in function), 12

G

Gamma() (built-in function), 12
GeneralizedExtremeValue() (built-in function), 12
GeneralizedPareto() (built-in function), 13
Geometric() (built-in function), 20
Gumbel() (built-in function), 13

H

Hypergeometric() (built-in function), 20

I

insupport() (built-in function), 24, 30, 50
invcov() (built-in function), 32
InverseGamma() (built-in function), 13
InverseGaussian() (built-in function), 14
invlogccdf() (built-in function), 24
invlogcdf() (built-in function), 24
isleptokurtic() (built-in function), 23
ismesokurtic() (built-in function), 23
isplatykurtic() (built-in function), 23

K

kurtosis() (built-in function), 23

L

Laplace() (built-in function), 14
length() (built-in function), 5, 29, 37, 41, 50
Levy() (built-in function), 14
location() (built-in function), 23, 34
logccdf() (built-in function), 24
logcdf() (built-in function), 24
logdetcov() (built-in function), 32
Logistic() (built-in function), 15
loglikelihood() (built-in function), 24, 30
LogNormal() (built-in function), 15
logpdf() (built-in function), 24, 30, 37, 41, 49

M

maximum() (built-in function), 50

mean() (built-in function), 23, 29, 37, 41
median() (built-in function), 23, 34
mgf() (built-in function), 24
minimum() (built-in function), 50
MixtureModel() (built-in function), 40
mode() (built-in function), 23, 34
modes() (built-in function), 23
MvNormal() (built-in function), 31, 32
MvNormalCanon() (built-in function), 33

N

ncategories() (built-in function), 23
NegativeBinomial() (built-in function), 21
Normal() (built-in function), 15
NormalInverseGaussian() (built-in function), 16
nsamples() (built-in function), 6
ntrials() (built-in function), 23

P

params() (built-in function), 22
Pareto() (built-in function), 16
pdf() (built-in function), 24, 30, 37, 41, 49
Poisson() (built-in function), 21
PoissonBinomial() (built-in function), 21
probs() (built-in function), 41

Q

quantile() (built-in function), 24, 49

R

rand() (built-in function), 6, 25, 30, 32, 37, 41, 49
rate() (built-in function), 23
Rayleigh() (built-in function), 16

S

sampler() (built-in function), 49, 50, 52
scale() (built-in function), 22, 34
shape() (built-in function), 23
size() (built-in function), 6, 29, 37, 52
Skellam() (built-in function), 22
skewness() (built-in function), 23
sqmahal() (built-in function), 32
std() (built-in function), 23
succprob() (built-in function), 22
SymTriangularDist() (built-in function), 17

T

TDist() (built-in function), 17
TriangularDist() (built-in function), 17

U

Uniform() (built-in function), 18

V

var() (built-in function), 23, 29, 41
VonMises() (built-in function), 18

W

Weibull() (built-in function), 18