Appendix for:

"Efficient correlation matching for fitting discrete multivariate distributions with arbitrary marginals and normal-copula dependence"

User's guide for Java classes with example

Overview

This document describes a Java implementation of correlation matching algorithms proposed in [1] for the situation where one wants to use the NORTA method to fit a multivariate distribution with discrete marginals. The four different algorithms discussed in [1] are implemented in four subclasses of an abstract class named `NortaInitDisc`. This software makes use of the SSJ library [2]. An example of how to use it is given at the end of this document.

The NORTA method is an approach for modeling dependence in a finite-dimensional random vector $X = (X_1, \ldots, X_d)$ with given univariate marginals via normal copula that fits the rank or the linear correlation between each pair of coordinates of X. The standard normal distribution function is applied to each coordinate of a vector $Z = (Z_1, \ldots, Z_d)$ of correlated standard normals to produce a vector $U = (U_1, \ldots, U_d)$ of correlated uniforms over $[0, 1]$. Then X is obtained by applying the inverse of each marginal distribution function to each coordinate of U. The fitting requires finding the correlation between the coordinates of each pair of Z that would yield the correlation between the coordinates of the corresponding pair of X. The step of finding the correlation matrix of Z, given the correlation matrix of X and the marginal distributions, constitutes the NORTA initialization step. In [1], we present a detailed analysis of the NORTA method and root-finding problem when the marginal distributions are discrete.

With the NORTA method, we have the following representation:

$$X_l = F_l^{-1}(\Phi(Z_l)), \quad l = 1, \ldots, d,$$

where Φ is the standard normal distribution function and $F_l^{-1}(u) = \inf\{x : F_l(x) \geq u\}$ for $0 \leq u \leq 1$, which is the quantile function of the marginal distribution $F_l, l = 1, \ldots, d$.

For the bivariate case ($d = 2$), we have a vector $X = (X_1, X_2)$ and the two marginal distributions F_1 and F_2 with means and standard deviations $\mu_{F_1} = E[F_1(X_1)]$, $\mu_{F_2} = E[F_2(X_2)]$, $\sigma_{F_1} = \mathrm{var}(F_1(X_1))^{1/2}$ and $\sigma_{F_2} = \mathrm{var}(F_2(X_2))^{1/2}$, respectively. For this case, NORTA initialization is reduced to the problem of finding the correlation $\rho_Z = \mathrm{Corr}(Z_1, Z_2)$.

In this document, we present a set of Java classes for NORTA initialization in the bivariate case given the rank correlation and two discrete marginal distributions. We have:

$$r_X(\rho) = \mathrm{Corr}(F_1(X_1), F_2(X_2)) = \frac{g_r(\rho) - \mu_{F_1}\mu_{F_2}}{\sigma_{F_1}\sigma_{F_2}}, \tag{1}$$

where:

$$\begin{aligned} g_r(\rho) &= E\left[F_1(X_1)F_2(X_2)\right] \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F_1\{F_1^{-1}[\Phi(x_1)]\}F_2\{F_2^{-1}[\Phi(x_2)]\}\phi_\rho(x_1,x_2)dx_1dx_2, \end{aligned} \tag{2}$$

where ϕ_ρ is the bivariate standard normal density. Then, for a given correlation r_X, we use an algorithm of root-finder to find the corresponding correlation ρ_Z that verifies

$$f_r(\rho_Z) = g_r(\rho_Z) - r_X\sigma_{F_1}\sigma_{F_2} - \mu_{F_1}\mu_{F_2} = 0. \tag{3}$$

When the marginal distributions are continuous, the root-finding problem is easy to solve when we use the rank correlation. We have an analytic solution for (2) and the relation in (1) becomes:

$$r_X(\rho) = (6/\pi)\arcsin(\rho/2).$$

NortaInitDisc

This abstract class defines the algorithms used for NORTA initialization when the marginal distributions are discrete. Four algorithms are supported for now, and they are defined as subclasses of the class `NortaInitDisc`.

For two random variables X_1 and X_2, and their two marginal distributions F_1 and F_2, respectively, we specify the rank correlation $r_X = \text{Corr}(F_1(X_1), F_2(X_2))$, the parameters of the marginal distributions and a parameter for truncation tr. For the correlation matching, we must have finite supports for the two distributions. Then if the support of each marginal is infinite, we have to upper-bound it at the quantile of order tr. For example, if the marginals have their support points in $[0, +\infty)$, the software will truncate to $[0, F_l^{-1}(tr)]$, for $l = 1, 2$. The parameter tr must to be given by the user, depending on the type of the two distributions. If the marginals have finite supports, one can simply give $tr = 1$.

Each algorithm `NI1, NI2a, NI2b and NI3` can be used to calculate the corresponding correlation $\rho_Z = \text{Corr}(Z_1, Z_2)$, where Z_1 and Z_2 are standard normal random variables. These subclasses implement the specific methods for NORTA initialization presented in [1].

Each type of algorithm should be defined as a subclass of `NortaInitDisc`. Each subclass must implement the method `computeCorr` which returns the solution ρ_Z. When executing this method, the subclass may call the methods `integ` and `deriv`, depending on the type of algorithm. For example, the subclass `NI1` calls only the method `integ`, since the algorithm do not use the derivative [1]. Each subclass must also call the method `computeParams` which is executed immediately before the beginning of the root-finder algorithm.

When creating a class representing an algorithm, the `toString` method can be called to display information about the inputs.

```
public abstract class NortaInitDisc
```

Constructor

```
public NortaInitDisc (double rX,
                      DiscreteDistributionInt dist1,
                      DiscreteDistributionInt dist2,
                      double tr)
```

Constructor with the target rank correlation `rX`, the two discrete marginals `dist1` and `dist2` and the parameter for the truncation `tr`. This constructor can be called only by the constructors of the subclasses.

Methods

`public abstract double computeCorr();`

Computes and returns the correlation ρ_Z. Every subclass of `NortaInitDisc` must implement this method.

`public void computeParams ()`

Computes the following inputs of each marginal distribution:

- The number of support points m_1 and m_2 for the two distributions.
- The means and standard deviations of $F_1(X_1)$ and $F_2(X_2)$, respectively.
- The vectors $p_1[i]$, $p_2[j]$, $z_1[i] = \Phi^{-1}(f_1[i])$ and $z_2[j] = \Phi^{-1}(f_2[j])$, where $f_1[i]$ and $f_2[j]$, for $i = 0, \ldots, m_1 - 1$; $j = 0, \ldots, m_2 - 1$, are the cumulative probability functions, and Φ is the standard normal distribution function.

Every subclass of `NortaInitDisc` must call this method.

`public double integ (double r)`

Computes the function

$$g_r(r) = \sum_{i=0}^{m_1-2} p_{1,i+1} \sum_{j=0}^{m_2-2} p_{2,j+1} \bar{\Phi}_r(z_{1,i}, z_{2,j}), \tag{4}$$

which involves the bivariate normal integral $\bar{\Phi}_r(x, y) = \int_x^\infty \int_y^\infty \phi_r(z_1, z_2) dz_1 dz_2$. Method `barF` of class `BiNormalDonnellyDist` (from package `probdistmulti` of SSJ [2]) is used to compute $\bar{\Phi}_r(x, y)$, with m_1, m_2, and the vectors $p_1[i], i = 1, \ldots, m_1-1$; $z_1[i], i = 0, \ldots, m_1-2$; $p_2[j], j = 1, \ldots, m_2 - 1$; $z_2[j], j = 0, \ldots, m_2 - 2$. The correlation parameter r must be in $[-1, 1]$. This method may be called by subclasses of `NortaInitDisc`.

`public double deriv (double r)`

Computes the derivative of g_r, given by

$$g_r'(r) = \sum_{i=0}^{m_1-2} p_{1,i+1} \sum_{j=0}^{m_2-2} p_{2,j+1} \phi_r(z_{1,i}, z_{2,j}), \tag{5}$$

where ϕ_r is the bivariate standard binormal density. The method uses m_1, m_2, and the vectors $p_1[i], i = 1, \ldots, m_1 - 1$; $z_1[i], i = 0, \ldots, m_1 - 2$; $p_2[j], j = 1, \ldots, m_2 - 1$; $z_2[j], j = 0, \ldots, m_2 - 2$. The correlation parameter r must be in $[-1, 1]$. This method may be called by subclasses of `NortaInitDisc`.

NI1

Extends the class `NortaInitDisc` and implements the algorithm NI1. It uses an algorithm based on Brent method for root-finding, which combines root-bracketing, bisection and inverse quadratic interpolation. It calls the method `integ` to compute the function g_r given in (4). The search should be done in the interval $[-1, 0]$ if $r_X \in [-1, 0]$, or $[0, 1]$ if $r_X \in [0, 1]$. At each iteration, the algorithm halves the interval length and uses an accuracy ϵ to find the root ρ_Z of equation (3).

```
public class NI1 extends NortaInitDisc
```

Constructor

```
public NI1 (double rX,
            DiscreteDistributionInt dist1,
            DiscreteDistributionInt dist2,
            double tr,
            double tolerance)
```

Constructor with the target rank correlation `rX`, the two discrete marginals `dist1` and `dist2`, the parameter for truncation `tr` (see the constructor of class `NortaInitDisc`) and the specific parameter $\epsilon =$ `tolerance` defined above for the algorithm NI1.

Methods

```
public double computeCorr ()
```

Computes and returns the correlation ρ_Z using the algorithm NI1.

NI2a

Extends the class `NortaInitDisc` and implements the algorithm NI2a. It uses the derivative, so it calls the method `deriv` to compute the function g'_r given in (5). The double integration in (2) is simplified and only a simple integration is used. The algorithm uses numerical integration with Simpson's rules over subintervals given by the finite sequence $\rho_k = \rho_0 + 2kh$, for $k = 0, 1, ..., m$, where h is a fixed step size and m is such that $1 - 2h < \rho_m < 1$. The initial point is chosen as $\rho_0 = 2\sin(\pi r_X/6)$. The integration is done between ρ_0 and $\rho_m = \pm(1-\delta)$, or between ρ_0 and 0, depending on the sign of r_X and on whether the root is to the left, or to the right of ρ_0. So depending on the case, the worst-case integration distance will be set to $d = |1 - \delta - \rho_0|$ or $d = |\rho_0|$. Then, the step size is readjusted to $h^* = d/(2m)$, where d is the maximum number of steps (iterations) calculated based on the pre-defined step size h, so $m = \lceil d/(2h) \rceil$. The algorithm stops at iteration k if the root is in a subinterval $[\rho_{k-1}, \rho_k]$, and a quadratic interpolation is used to compute the solution. For this, the method `interpol` of class `Misc` (from package `util` of SSJ [2]) is used.

```
public class NI2a extends NortaInitDisc
```

Constructor

```
public NI2a(double rX,
            DiscreteDistributionInt dist1,
            DiscreteDistributionInt dist2,
            double tr,
            double h,
            double delta)
```

Constructor with the target rank correlation `rX`, the two discrete marginals `dist1` and `dist2`, the paramater for the truncation `tr` (see the constructor of class `NortaInitDisc`), and the specific parameters `h` and $\delta =$ `delta` for the algorithm NI2a, as described above.

Methods

```
public double computeCorr ()
```

Computes and returns the correlation ρ_Z using the algorithm NI2a.

NI2b

Extends the class `NortaInitDisc` and implements the algorithm NI2b. It is a variant of NI2a. It uses the derivative, so it calls the method `deriv` to compute the function g'_r given in (5) and uses numerical integration with Simpson's rules as well. But the integration grid is either in the interval $[0, 1-\delta]$ or $[-1+\delta, 0]$, depending on the sign of r_X. Here the number of subintervals of integration is fixed to m and the algorithm stops at iteration k if the root is in subinterval $[\rho_{k-1}, \rho_k]$, and a quadratic interpolation is used to compute the solution. For this, the method `interpol` of class `Misc` (from package `util` of SSJ [2]) is used.

```
public class NI2b extends NortaInitDisc
```

Constructor

```
public NI2b(double rX,
            DiscreteDistributionInt dist1,
            DiscreteDistributionInt dist2,
            double tr,
            int m,
            double delta)
```

Constructor with the target rank correlation `rX`, the two discrete marginals `dist1` and `dist2`, the parameter for the truncation `tr` (see the constructor of class `NortaInitDisc`), and the specific parameters `m` and $\delta =$ `delta` for the algorithm NI2b, as described above.

Methods

```
public double computeCorr ()
```

Computes and returns the correlation ρ_Z using the algorithm NI2b.

NI3

Extends the class `NortaInitDisc` and implements the algorithm NI3. It uses the function g_r and its derivative g'_r, so it calls the methods `integ` and `deriv`, given in (4) and (5) and uses an adapted version of the Newton-Raphson algorithm. At any iteration, if the solution falls outside the search interval, the algorithm uses bisection and halves the interval length to guarantee convergence. The initial solution is taken as $\rho_0 = 2\sin(\pi r_X/6)$, and then at each iteration k, $f_r(\rho_k)$ and $f'_r(\rho_k)$ are calculated and a solution is computed by the recurrence formula:

$$\rho_{k+1} = \rho_k - \frac{f_r(\rho_k)}{f'_r(\rho_k)}.$$

The algorithm stops at iteration k if $|\rho_{k-1} - \rho_k| \leq \epsilon$. The function f_r is the one given in (3).

```
public class NI3 extends NortaInitDisc
```

Constructor

```
public NI3 (double rX,
            DiscreteDistributionInt dist1,
            DiscreteDistributionInt dist2,
            double tr,
            double tolerance)
```

Constructor with the target rank correlation `rX`, the two discrete marginals `dist1` and `dist2`, the parameter for the truncation `tr` (see the constructor of class `NortaInitDisc`), and the specific parameter $\epsilon =$ `tolerance` for the algorithm NI3, as defined above.

Methods

```
public double computeCorr ()
```

Computes and returns the correlation ρ_Z using algorithm NI3.

Example

In this example, we consider two random variables X_1 and X_2 with negative binomial marginals, denoted by NegBin(s, p). In our example, the parameters (s, p) for X_1 and X_2, respectively, are: $s_1 = 15.68$, $p_1 = 0.3861$, $s_2 = 60.21$ and $p_2 = 0.6211$. We want to calculate the correlation ρ_Z for a target rank correlation $r_X = 0.43$. Since the negative binomial has an unbounded support, we set the upper bound points of each support at the quantile of order $tr = 1 - 10^{-6}$, so that the number of support points $m_l = F_l^{-1}(1 - 10^{-6}) + 1$, for $l = 1, 2$.

The Java program uses the class `DiscreteDistributionInt` of package `probdist` from SSJ, to specify the two discrete marginal distributions. Each of the four subclasses `NI1, NI2a, NI2b and NI3` are called for each algorithm to compute the correlation ρ_Z, so we can compare the results.

Listing 1: Example with correlated negative binomial distributions.

```
import umontreal.iro.lecuyer.probdist.*;

public class ExampleNortaInitDisc
{
    public static void main (String[] args) {
        final double rX = 0.43;              // Target rank correlation rX
        final double tr = 1.0 - 1.0e-6;      // Quantile upper limit

        // Define the two marginal distributions
        DiscreteDistributionInt dist1 = new NegativeBinomialDist(15.68, 0.3861);
        DiscreteDistributionInt dist2 = new NegativeBinomialDist(60.21, 0.6211);

        NI1 ni1Obj = new NI1(rX, dist1, dist2, tr, 1.0e-4);
        System.out.println("Result with method NI1:  rho_Z = "
                           + ni1Obj.computeCorr());
        NI2a ni2aObj = new NI2a(rX, dist1, dist2, tr, 0.005, 1.0e-4);
        System.out.println("Result with method NI2b: rho_Z = "
                           + ni2aObj.computeCorr());
        NI2b ni2bObj = new NI2b(rX, dist1, dist2, tr, 5, 1.0e-4);
        System.out.println("Result with method NI2a: rho_Z = "
                           + ni2bObj.computeCorr());
        NI3 ni3Obj = new NI3(rX, dist1, dist2, tr, 1.0e-4);
        System.out.println("Result with method NI3:  rho_Z = "
                           + ni3Obj.computeCorr());
    }
}
```

Listing 2: Results of the program `ExampleNortaInitDisc.java`

```
Result with method NI1:  rho_Z = 0.44691
Result with method NI2b: rho_Z = 0.44685
Result with method NI2a: rho_Z = 0.44683
Result with method NI3:  rho_Z = 0.44691
```

References

[1] A. N. Avramidis, N. Channouf, and P. L'Ecuyer. Efficient correlation matching for fitting discrete multivariate distributions with arbitrary marginals and normal-copula dependence. Submitted, 2006.

[2] P. L'Ecuyer. *SSJ: A Java Library for Stochastic Simulation*, 2004. Software user's guide, Disponible `http://www.iro.umontreal.ca/~lecuyer`.