JuliaExamples

Examples of using the toolkit in Julia

julia, examples, Phase-Deploy

Updated Sep 9, 2014 by joseph.lizier

Demos > Julia code examples

Julia code examples

This page describes a basic set of demonstration scripts for using the toolkit in Julia. The .jl files can be found at demos/julia in the svn or main distributions (from the V1.1 release at a future point in time ...).

Please see UseInJulia for instructions on how to begin using JIDT from inside Julia, using the `JavaCall` package.

This page contains the following code examples:

- Example 1 - Transfer entropy on binary data
- Example 3 - Transfer entropy on continuous data using kernel estimators
- Example 4 - Transfer entropy on continuous data using Kraskov estimators
- Example 6 - Dynamic dispatch with Mutual info calculator

Be aware that multidimensional arrays cannot yet be passed from Julia to Java via the `JavaCall` package; as such we have not implemented Example 2 nor Example 5 from the SimpleJavaExamples and Example 6 is only implemented up to the point at which the multidimensional data is passed to Java.

Example 1 - Transfer entropy on binary data

example1TeBinaryData.jl - Simple transfer entropy (TE) calculation on binary data using the discrete TE calculator:

```
# Import the JavaCall package:
using JavaCall;

# Change location of jar to match yours:
jarLocation = "../../infodynamics.jar";
# Start the JVM supplying classpath and heap size
#  (increase memory here if you get crashes due to not enough space)
JavaCall.init(["-Djava.class.path=$(jarLocation)", "-Xmx128M"]);

# Generate some random binary data.
sourceArray=rand(0:1, 100);
destArray = [0; sourceArray[1:99]];
sourceArray2=rand(0:1, 100);

# Create a TE calculator and run it:
teClass = @jimport infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete;
teCalc=teClass((jint, jint), 2, 1);
jcall(teCalc, "initialise", Void, ()); # This is how to indicate a void return type and arguments

# We can pass simple arrays of ints directly in:
jcall(teCalc, "addObservations", Void, (Array{jint,1}, Array{jint,1}),
                                     sourceArray, destArray);
result = jcall(teCalc, "computeAverageLocalOfObservations", jdouble, ());
@printf("For copied source, result should be close to 1 bit : %.4f\n", result);

jcall(teCalc, "initialise", Void, ());
jcall(teCalc, "addObservations", Void, (Array{jint,1}, Array{jint,1}),
                                     sourceArray2, destArray);
result2 = jcall(teCalc, "computeAverageLocalOfObservations", jdouble, ());
@printf("For random source, result should be close to 0 bits: %.4f\n", result2);
```

Example 3 - Transfer entropy on continuous data using kernel estimators

example3TeContinuousDataKernel.jl - Simple transfer entropy (TE) calculation on continuous-valued data using the (box) kernel-estimator TE calculator.

```
# Import the JavaCall package:
```

```
using JavaCall;

# Change location of jar to match yours:
jarLocation = "../../infodynamics.jar";
# Start the JVM supplying classpath and heap size
#  (increase memory here if you get crashes due to not enough space)
JavaCall.init(["-Djava.class.path=$(jarLocation)", "-Xmx128M"]);

# Generate some random normalised data.
numObservations = 1000;
covariance=0.4;
sourceArray=randn(numObservations);
destArray = [0, covariance*sourceArray[1:numObservations-1] + (1-covariance)*randn(numObservations - 1)];
sourceArray2=randn(numObservations); # Uncorrelated source
# Create a TE calculator and run it:
teClass = @jimport infodynamics.measures.continuous.kernel.TransferEntropyCalculatorKernel;
teCalc=teClass(());
jcall(teCalc, "setProperty", Void, (JString, JString), "NORMALISE", "true"); # Normalise the individual variables
jcall(teCalc, "initialise", Void, (jint, jdouble), 1, 0.5); # Use history length 1 (Schreiber k=1), kernel width of 0
jcall(teCalc, "setObservations", Void, (Array{jdouble,1}, Array{jdouble,1}),
                                 sourceArray, destArray);
# For copied source, should give something close to expected value for correlated Gaussians:
result = jcall(teCalc, "computeAverageLocalOfObservations", jdouble, ());
@printf("TE result %.4f bits; expected to be close to %.4f bits for these correlated Gaussians but biased upwards\n",

jcall(teCalc, "initialise", Void, ()); # Initialise leaving the parameters the same
jcall(teCalc, "setObservations", Void, (Array{jdouble,1}, Array{jdouble,1}),
                                 sourceArray2, destArray);
# For random source, it should give something close to 0 bits
result2 = jcall(teCalc, "computeAverageLocalOfObservations", jdouble, ());
@printf("TE result %.4f bits; expected to be close to 0 bits for uncorrelated Gaussians but will be biased upwards\n"

# We can get insight into the bias by examining the null distribution:
empiricalDistClass = @jimport infodynamics.utils.EmpiricalMeasurementDistribution;
nullDist = jcall(teCalc, "computeSignificance", empiricalDistClass, (jint,), 100);
@printf("Null distribution for unrelated source and destination (i.e. the bias) has mean %.4f and standard deviation
        jcall(nullDist, "getMeanOfDistribution", jdouble, ()),
        jcall(nullDist, "getStdOfDistribution", jdouble, ()));
```

Example 4 - Transfer entropy on continuous data using Kraskov estimators

example4TeContinuousDataKraskov.jl - Simple transfer entropy (TE) calculation on continuous-valued data using the Kraskov-estimator TE calculator.

```
# Import the JavaCall package:
using JavaCall;

# Change location of jar to match yours:
jarLocation = "../../infodynamics.jar";
# Start the JVM supplying classpath and heap size
#  (increase memory here if you get crashes due to not enough space)
JavaCall.init(["-Djava.class.path=$(jarLocation)", "-Xmx128M"]);

# Generate some random normalised data.
numObservations = 1000;
covariance=0.4;
sourceArray=randn(numObservations);
destArray = [0, covariance*sourceArray[1:numObservations-1] + (1-covariance)*randn(numObservations - 1)];
sourceArray2=randn(numObservations); # Uncorrelated source

# Create a TE calculator and run it:
teClass = @jimport infodynamics.measures.continuous.kraskov.TransferEntropyCalculatorKraskov;
teCalc=teClass(());
jcall(teCalc, "setProperty", Void, (JString, JString), "k", "4"); # Use Kraskov parameter K=4 for 4 nearest points
jcall(teCalc, "initialise", Void, (jint, ), 1); # Use history length 1 (Schreiber k=1)
# Perform calculation with correlated source:
jcall(teCalc, "setObservations", Void, (Array{jdouble,1}, Array{jdouble,1}),
                                 sourceArray, destArray);
result = jcall(teCalc, "computeAverageLocalOfObservations", jdouble, ());
# Note that the calculation is a random variable (because the generated
#  data is a set of random variables) - the result will be of the order
#  of what we expect, but not exactly equal to it; in fact, there will
#  be a large variance around it.
@printf("TE result %.4f nats; expected to be close to %.4f nats for these correlated Gaussians\n",
    result, log(1/(1-covariance^2)));

# Perform calculation with uncorrelated source:
jcall(teCalc, "initialise", Void, ()); # Initialise leaving the parameters the same
jcall(teCalc, "setObservations", Void, (Array{jdouble,1}, Array{jdouble,1}),
                                 sourceArray2, destArray);
result2 = jcall(teCalc, "computeAverageLocalOfObservations", jdouble, ());
@printf("TE result %.4f nats; expected to be close to 0 nats for these uncorrelated Gaussians\n", result2);

# We can also compute the local TE values for the time-series samples here:
```

```
#  (See more about utility of local TE in the CA demos)
localTE = jcall(teCalc, "computeLocalOfPreviousObservations", Array{jdouble,1}, ());
@printf("Notice that the mean of locals, %.4f nats, equals the previous result\n",
        sum(localTE)/(numObservations-1));
```

Example 6 - Dynamic dispatch with Mutual info calculator

example6DynamicCallingMutualInfo.jl - This example shows how to write Julia code to take advantage of the common interfaces defined for various information-theoretic calculators. Here, we use the common form of the `infodynamics.measures.continuous.MutualInfoCalculatorMultiVariate` interface (which is never named here) to write common code into which we can plug one of three concrete implementations (kernel estimator, Kraskov estimator or linear-Gaussian estimator) by dynamically supplying the class name of the concrete implementation.

While the dynamic dispatch works here, we can't run all of the functionality of this example as would be seen in Java or other languages, since it requires multidimensional arrays to be passed between Julia and Java which is not yet supported in `JavaCall` (see UseInJulia).

```
# Import the JavaCall package:
using JavaCall;

# Change location of jar to match yours:
jarLocation = "../../infodynamics.jar";
# Start the JVM supplying classpath and heap size
#  (increase memory here if you get crashes due to not enough space)
JavaCall.init(["-Djava.class.path=$(jarLocation)", "-Xmx128M"]);

#---------------------
# 1. Properties for the calculation (these are dynamically changeable, you could
#    load them in from another properties file):
# The name of the data file (relative to this directory)
datafile = "../data/4ColsPairedNoisyDependence-1.txt";
# List of column numbers for variables 1 and 2:
#  (you can select any columns you wish to be contained in each variable)
variable1Columns = [1,2]; # array indices start from 1 in Julia
variable2Columns = [3,4];
# The name of the concrete implementation of the interface
#  infodynamics.measures.continuous.MutualInfoCalculatorMultiVariate
#  which we wish to use for the calculation.
# Note that one could use any of the following calculators (try them all!):
#  implementingClass = "infodynamics.measures.continuous.kraskov.MutualInfoCalculatorMultiVariateKraskov1"; % MI([1,2
#  implementingClass = "infodynamics.measures.continuous.kernel.MutualInfoCalculatorMultiVariateKernel";
#  implementingClass = "infodynamics.measures.continuous.gaussian.MutualInfoCalculatorMultiVariateGaussian";
implementingClass = "infodynamics.measures.continuous.kraskov.MutualInfoCalculatorMultiVariateKraskov1";

#---------------------
# 2. Load in the data
data = readdlm(datafile, ' ', '\n')
# Pull out the columns from the data set which correspond to each of variable 1 and 2:
variable1 = data[:, variable1Columns];
variable2 = data[:, variable2Columns];

#---------------------
# 3. Dynamically instantiate an object of the given class:
# Since @jimport uses the provided text directly (rather than evaluating a variable)
#  we need to do some Julia interpolation of $implementingClass to get our
#  dynamic reference to the class.
miCalcClass = eval(:(@jimport $implementingClass));
miCalc = miCalcClass(());

#---------------------
# 4. Start using the MI calculator, paying attention to only
#  call common methods defined in the interface type
#  infodynamics.measures.continuous.MutualInfoCalculatorMultiVariate
#  not methods only defined in a given implementation class.
# a. Initialise the calculator to use the required number of
#   dimensions for each variable:
jcall(miCalc, "initialise", Void, (jint,jint), length(variable1Columns), length(variable2Columns));
# b. Supply the observations to compute the PDFs from:

# For the moment we have to exit this example here::
@printf("We're stuck at this point until support for multidimensional arrays is included in JavaCall");
exit();
```