SimpleJavaExamples

Joseph Lizier edited this page on 20 Aug 2015 · 2 revisions

Simple examples of using the toolkit

Demos > Simple Java code examples

Simple Java code examples

This page describes a basic set of demonstration scripts for using the toolkit. The .java files can be found at demos/java/infodynamics/demos in the svn or main distributions. The demos can be compiled and run with the relevant shell scripts in demos/java (one for each example, e.g. example1TeBinaryData.sh) Please note that other more complicated examples are available from the main Demos page.

This page contains the following code examples:

Example 1 - Transfer entropy on binary data

Example1TeBinaryData.java - Simple transfer entropy (TE) calculation on binary data using the discrete TE calculator:

```
// Requires the following imports before the class definition:
```

```
// import infodynamics.utils.RandomGenerator;
// import infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete;

int arrayLengths = 100;
RandomGenerator rg = new RandomGenerator();

// Generate some random binary data:
int[] sourceArray = rg.generateRandomInts(arrayLengths, 2);
int[] destArray = new int[arrayLengths];
destArray[0] = 0;
System.arraycopy(sourceArray, 0, destArray, 1, arrayLengths - 1);
int[] sourceArray2 = rg.generateRandomInts(arrayLengths, 2);

// Create a TE calculator and run it:
TransferEntropyCalculatorDiscrete teCalc=
        new TransferEntropyCalculatorDiscrete(2, 1);
teCalc.initialise();
teCalc.addObservations(sourceArray, destArray);
double result = teCalc.computeAverageLocalOfObservations();
System.out.printf("For copied source, result should be close to 1 bit : %.3f bits\n"
teCalc.initialise();
teCalc.addObservations(sourceArray2, destArray);
double result2 = teCalc.computeAverageLocalOfObservations();
System.out.printf("For random source, result should be close to 0 bits: %.3f bits\n"
```

Example 2 - Transfer entropy on multidimensional binary data

Example2TeMultidimBinaryData.java - Simple transfer entropy (TE) calculation on multidimensional binary data using the discrete TE calculator.

This example shows how to handle multidimensional arrays where we pool the observations over all variables with the discrete calculator.

```
// Requires the following imports before the class definition:
// import infodynamics.utils.RandomGenerator;
// import infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete;

int timeSteps = 2;
int variables = 100;
RandomGenerator rg = new RandomGenerator();

// Create many columns in a multidimensional array (2 rows by 100 columns),
//  where the next time step (row 2) copies the value of the column on the left
//  from the previous time step (row 1):
int[][] twoDTimeSeries = new int[timeSteps][];
twoDTimeSeries[0] = rg.generateRandomInts(variables, 2);
twoDTimeSeries[1] = new int[variables];
twoDTimeSeries[1][0] = twoDTimeSeries[0][variables - 1];
```

```
System.arraycopy(twoDTimeSeries[0], 0, twoDTimeSeries[1], 1, variables - 1);

// Create a TE calculator and run it:
TransferEntropyCalculatorDiscrete teCalc=
        new TransferEntropyCalculatorDiscrete(2, 1);
teCalc.initialise();
// Add observations of transfer across one cell to the right (j=1)
//  per time step:
teCalc.addObservations(twoDTimeSeries, 1);

double result2D = teCalc.computeAverageLocalOfObservations();
System.out.printf("The result should be close to 1 bit here, " +
        "since we are executing copy operations of what is effectively " +
        "a random bit to each cell here: %.3f bits\n", result2D);
```

Example 3 - Transfer entropy on continuous data using kernel estimators

Example3TeContinuousDataKernel.java - Simple transfer entropy (TE) calculation on continuous-valued data using the (box) kernel-estimator TE calculator.

```
// Requires the following imports before the class definition:
// import infodynamics.utils.RandomGenerator;
// import infodynamics.measures.continuous.kernel.TransferEntropyCalculatorKernel;

// Generate some random normalised data.
int numObservations = 1000;
double covariance = 0.4;

// Create destArray correlated to previous value of sourceArray:
RandomGenerator rg = new RandomGenerator();
double[] sourceArray = rg.generateNormalData(numObservations, 0, 1);
double[] destArray = rg.generateNormalData(numObservations, 0, 1-covariance);
for (int t = 1; t < numObservations; t++) {
    destArray[t] += covariance * sourceArray[t-1];
}
// And an uncorrelated second source
double[] sourceArray2 = rg.generateNormalData(numObservations, 0, 1);

// Create a TE calculator and run it:
TransferEntropyCalculatorKernel teCalc =
        new TransferEntropyCalculatorKernel();
teCalc.setProperty("NORMALISE", "true"); // Normalise the individual variables (defa
teCalc.initialise(1, 0.5); // Use history length 1 (Schreiber k=1), kernel width of
teCalc.setObservations(sourceArray, destArray);
// For copied source, should give something close to 1 bit:
double result = teCalc.computeAverageLocalOfObservations();
System.out.printf("TE result %.4f bits; expected to be close to " +
        "%.4f bits for these correlated Gaussians but biased upwards\n",
```

```
        result, Math.log(1.0/(1-Math.pow(covariance,2)))/Math.log(2));

teCalc.initialise(); // Initialise leaving the parameters the same
teCalc.setObservations(sourceArray2, destArray);
// For random source, it should give something close to 0 bits
double result2 = teCalc.computeAverageLocalOfObservations();
System.out.printf("TE result %.4f bits; expected to be close to " +
        "0 bits for uncorrelated Gaussians but will be biased upwards\n",
        result2);

// We can get insight into the bias by examining the null distribution:
EmpiricalMeasurementDistribution nullDist = teCalc.computeSignificance(100);
System.out.printf("Null distribution for unrelated source and destination " +
    "(i.e. the bias) has mean %.4f and standard deviation %.4f\n",
    nullDist.getMeanOfDistribution(), nullDist.getStdOfDistribution());
```

Example 4 - Transfer entropy on continuous data using Kraskov estimators

Example4TeContinuousDataKraskov.java - Simple transfer entropy (TE) calculation on continuous-valued data using the Kraskov-estimator TE calculator.

```
// Requires the following imports before the class definition:
// import infodynamics.utils.RandomGenerator;
// import infodynamics.measures.continuous.kraskov.TransferEntropyCalculatorKraskov;

// Generate some random normalised data.
int numObservations = 1000;
double covariance = 0.4;

// Create destArray correlated to previous value of sourceArray:
RandomGenerator rg = new RandomGenerator();
double[] sourceArray = rg.generateNormalData(numObservations, 0, 1);
double[] destArray = rg.generateNormalData(numObservations, 0, 1-covariance);
for (int t = 1; t < numObservations; t++) {
    destArray[t] += covariance * sourceArray[t-1];
}
// And an uncorrelated second source
double[] sourceArray2 = rg.generateNormalData(numObservations, 0, 1);

// Create a TE calculator and run it:
TransferEntropyCalculatorKraskov teCalc =
        new TransferEntropyCalculatorKraskov();
teCalc.setProperty("k", "4"); // Use Kraskov parameter K=4 for 4 nearest neighbours
teCalc.initialise(1); // Use history length 1 (Schreiber k=1)

// Perform calculation with correlated source:
teCalc.setObservations(sourceArray, destArray);
double result = teCalc.computeAverageLocalOfObservations();
```

```
// 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.
System.out.printf("TE result %.4f nats; expected to be close to " +
        "%.4f nats for these correlated Gaussians\n",
        result, Math.log(1.0/(1-Math.pow(covariance,2))));

//  Perform calculation with uncorrelated source:
teCalc.initialise(); // Initialise leaving the parameters the same
teCalc.setObservations(sourceArray2, destArray);
// For random source, it should give something close to 0 bits
double result2 = teCalc.computeAverageLocalOfObservations();
System.out.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)
double[] localTE = teCalc.computeLocalOfPreviousObservations();
System.out.printf("Notice that the mean of locals, %.4f nats," +
    " equals the previous result\n",
    MatrixUtils.sum(localTE)/(double)(numObservations-1));
```

Example 5 - Multivariate transfer entropy on binary data

Example5TeBinaryMultivarTransfer.java - Multivariate transfer entropy (TE) calculation on binary data using the discrete TE calculator.

```
// Requires the following imports before the class definition:
// import infodynamics.utils.MatrixUtils;
// import infodynamics.utils.RandomGenerator;
// import infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete;

// Generate some random binary data.
int timeSeriesLength = 100;
RandomGenerator rg = new RandomGenerator();
int[][] sourceArray = rg.generateRandomInts(timeSeriesLength, 2, 2);
int[][] sourceArray2 = rg.generateRandomInts(timeSeriesLength, 2, 2);
// Destination variable takes a copy of the first bit of the
//  previous source value in bit 0,
//  and an XOR of the two previous bits of the source in bit 1:
int[][] destArray = new int[timeSeriesLength][2];
for (int r = 1; r < timeSeriesLength; r++) {
    // This is a bitwise XOR, but is fine for our purposes
    //  with binary data:
    destArray[r][0] = sourceArray[r - 1][0];
    destArray[r][1] = sourceArray[r - 1][0] ^ sourceArray[r - 1][1];
}

// Create a TE calculator and run it.
```

```
// Need to represent 4-state variables for the joint destination variable
TransferEntropyCalculatorDiscrete teCalc=
        new TransferEntropyCalculatorDiscrete(4, 1);
teCalc.initialise();

// We need to construct the joint values of the dest and source before we pass them
teCalc.addObservations(MatrixUtils.computeCombinedValues(sourceArray, 2),
        MatrixUtils.computeCombinedValues(destArray, 2));

double result = teCalc.computeAverageLocalOfObservations();
System.out.printf("For source which the 2 bits are determined from, " +
        "result should be close to 2 bits : %.3f\n", result);

// Check random source:
teCalc.initialise();
teCalc.addObservations(MatrixUtils.computeCombinedValues(sourceArray2, 2),
        MatrixUtils.computeCombinedValues(destArray, 2));
double result2 = teCalc.computeAverageLocalOfObservations();
System.out.printf("For random source, result should be close to 0 bits " +
        "in theory: %.3f\n", result2);

System.out.printf("The result for random source is inflated towards 0.3 " +
        " due to finite observation length (%d). One can verify that the " +
        "answer is consistent with that from a random source by checking: " +
        "teCalc.computeSignificance(1000); ans.pValue\n",
        teCalc.getNumObservations());
```

Example 6 - Late binding Mutual info calculator

Example6LateBindingMutualInfo.java - This class is used to demonstrate the manner in which a user can code to the interfaces defined in `infodynamics.measures.continuous` , and dynamically alter the instantiated class at runtime. We demonstrate this using a multivariate mutual information calculation. The properties file (supplied on command line) to specify the name of the dynamically instantiated class is available in the distribution at demos/java /example6LateBindingMutualInfo.props.

This example also demonstrates how to read simple files of arrays of data with the toolkit, as well as how to dynamically load properties from a java properties file.

```
// Requires the following imports before the class definition:
// import infodynamics.measures.continuous.MutualInfoCalculatorMultiVariate;
// import infodynamics.utils.ArrayFileReader;
// import infodynamics.utils.MatrixUtils;
// import infodynamics.utils.ParsedProperties;

/**
 * @param args One command line argument taken, specifying location of
 *  the properties file. This should be example6LateBindingMutualInfo.props
```

```
     *  in the demos/java directory.
     */
    public static void main(String[] args) throws Exception {

        // 0. Preliminaries (reading in the dynamic properties and the data):
        //  a. Read in the properties file defined as the first
        //     command line argument:
        ParsedProperties props = new ParsedProperties(args[0]);
        //  b. Read in the data file, whose filename is defined in the
        //     property "datafile" in our properties file:
        ArrayFileReader afr = new ArrayFileReader(props.getStringProperty("datafile"));
        double[][] data = afr.getDouble2DMatrix();
        //  c. Pull out the columns from the data set which
        //     correspond to the univariate and joint variables we will work with:
        //     First the univariate series to compute standard MI between:
        int univariateSeries1Column = props.getIntProperty("univariateSeries1Column");
        int univariateSeries2Column = props.getIntProperty("univariateSeries2Column");
        double[] univariateSeries1 = MatrixUtils.selectColumn(data, univariateSeries1Col
        double[] univariateSeries2 = MatrixUtils.selectColumn(data, univariateSeries2Col
        //     Next the multivariate series to compute joint or multivariate MI between:
        int[] jointVariable1Columns = props.getIntArrayProperty("jointVariable1Columns")
        int[] jointVariable2Columns = props.getIntArrayProperty("jointVariable2Columns")
        double[][] jointVariable1 = MatrixUtils.selectColumns(data, jointVariable1Column
        double[][] jointVariable2 = MatrixUtils.selectColumns(data, jointVariable2Column

        // 1. Create a reference for our calculator as
        //  an object implementing the interface type:
        MutualInfoCalculatorMultiVariate miCalc;

        // 2. Define the name of the class to be instantiated here:
        String implementingClass = props.getStringProperty("implementingClass");

        // 3. Dynamically instantiate an object of the given class:
        //  Part 1: Class.forName(implementingClass) grabs a reference to
        //   the class named by implementingClass.
        //  Part 2: .newInstance() creates an object instance of that class.
        //  Part 3: (MutualInfoCalculatorMultiVariate) casts the return
        //   object into an instance of our generic interface type.
        miCalc = (MutualInfoCalculatorMultiVariate)
                Class.forName(implementingClass).newInstance();

        // 4. Start using our MI calculator, paying attention to only
        //  call common methods defined in the interface type, not methods
        //  only defined in a given implementation class.
        // a. Initialise the calculator for a univariate calculation:
        miCalc.initialise(1, 1);
        // b. Supply the observations to compute the PDFs from:
        miCalc.setObservations(univariateSeries1, univariateSeries2);
        // c. Make the MI calculation:
        double miUnivariateValue = miCalc.computeAverageLocalOfObservations();

        // 5. Continue onto a multivariate calculation, still only
```

```
        //  calling common methods defined in the interface type.
        // a. Initialise the calculator for a multivariate calculation
        //  to use the required number of dimensions for each variable:
        miCalc.initialise(jointVariable1Columns.length, jointVariable2Columns.length);
        // b. Supply the observations to compute the PDFs from:
        miCalc.setObservations(jointVariable1, jointVariable2);
        // c. Make the MI calculation:
        double miJointValue = miCalc.computeAverageLocalOfObservations();

        System.out.printf("MI calculator %s computed the univariate MI(%d;%d) as %.5f "
                " and joint MI as %.5f\n",
                implementingClass, univariateSeries1Column, univariateSeries2Column,
                miUnivariateValue, miJointValue);
    }
```

Example 7 - Ensemble approach for multiple samples

Example7EnsembleMethodTeContinuousDataKraskov.java - This class is used to demonstrate the manner in which a user supplies an ensemble of samples from multiple time series.

```
// Requires the following imports before the class definition:
// import infodynamics.measures.continuous.kraskov.TransferEntropyCalculatorKraskov;
// import infodynamics.utils.RandomGenerator;

// Prepare to generate some random normalised data.
int numObservations = 1000;
double covariance = 0.4;
RandomGenerator rg = new RandomGenerator();

// Create a TE calculator and run it:
TransferEntropyCalculatorKraskov teCalc =
        new TransferEntropyCalculatorKraskov();
teCalc.setProperty("k", "4"); // Use Kraskov parameter K=4 for 4 nearest neighbours
teCalc.initialise(1); // Use history length 1 (Schreiber k=1)
teCalc.startAddObservations();

for (int trial = 0; trial < 10; trial++) {

    // Create a new trial, with destArray correlated to
    //  previous value of sourceArray:
    double[] sourceArray = rg.generateNormalData(numObservations, 0, 1);
    double[] destArray = rg.generateNormalData(numObservations, 0, 1-covariance);
    for (int t = 1; t < numObservations; t++) {
        destArray[t] += covariance * sourceArray[t-1];
    }

    // Add observations for this trial:
    System.out.printf("Adding samples from trial %d ...\n", trial);
    teCalc.addObservations(sourceArray, destArray);
}
```

```
// We've finished adding trials:
System.out.println("Finished adding trials");
teCalc.finaliseAddObservations();

// Compute the result:
System.out.println("Computing TE ...");
double result = teCalc.computeAverageLocalOfObservations();
// 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 some variance around it (smaller than example 4 since we have more samples).
System.out.printf("TE result %.4f nats; expected to be close to " +
        "%.4f nats for these correlated Gaussians\n",
        result, Math.log(1.0/(1-Math.pow(covariance,2))));
```

Example 8 - Transfer entropy on continuous data using binning

Example8TeContinuousDataByBinning.java - Simple transfer entropy (TE) calculation on continuous-valued data by binning the continuous data to discrete, then using a discrete TE calculator.

```
// Requires the following imports before the class definition:
// import infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete;
// import infodynamics.utils.MatrixUtils;
// import infodynamics.utils.RandomGenerator;

// Prepare to generate some random normalised data.
int numObservations = 1000;
double covariance = 0.4;
int numDiscreteLevels = 4;

// Create destArray correlated to previous value of sourceArray:
RandomGenerator rg = new RandomGenerator();
double[] sourceArray = rg.generateNormalData(numObservations, 0, 1);
double[] destArray = rg.generateNormalData(numObservations, 0, 1-covariance);
for (int t = 1; t < numObservations; t++) {
    destArray[t] += covariance * sourceArray[t-1];
}

// Discretize or bin the data -- one could also call:
//  MatrixUtils.discretiseMaxEntropy for a maximum entropy binning
int[] binnedSource = MatrixUtils.discretise(sourceArray, numDiscreteLevels);
int[] binnedDest = MatrixUtils.discretise(destArray, numDiscreteLevels);

// Create a TE calculator and run it:
TransferEntropyCalculatorDiscrete teCalc =
    new TransferEntropyCalculatorDiscrete(numDiscreteLevels, 1);
```

```
teCalc.initialise();
teCalc.addObservations(binnedSource, binnedDest);
double result = teCalc.computeAverageLocalOfObservations();
// Calculation will be heavily biased because of the binning,
//  and the small number of samples
System.out.printf("TE result %.4f bits; expected to be close to " +
        "%.4f bits for these correlated Gaussians\n",
        result, Math.log(1.0/(1-Math.pow(covariance,2)))/Math.log(2));
```

Example 9 - Transfer entropy on continuous data using Kraskov estimators with auto-embedding

Example9TeKraskovAutoEmbedding.java - Transfer entropy (TE) calculation on continuous-valued data using the Kraskov-estimator TE calculator, with automatic selection of embedding parameters via the Ragwitz criteria.

Available from release v1.3 (in future) or currently available via SVN

```
// Requires the following imports before the class definition:
// import infodynamics.measures.continuous.kraskov.TransferEntropyCalculatorKraskov;
// import infodynamics.utils.MatrixUtils;
// import infodynamics.utils.ArrayFileReader;

// Examine the heart-breath interaction that Schreiber originally looked at:
ArrayFileReader afr = new ArrayFileReader("../data/SFI-heartRate_breathVol_bloodOx.t
double[][] data = afr.getDouble2DMatrix();
// Select data points 2350:3550
data = MatrixUtils.selectRows(data, 2349, 3550-2350+1);

// Create a Kraskov TE calculator:
TransferEntropyCalculatorKraskov teCalc = new TransferEntropyCalculatorKraskov();

// Set properties for auto-embedding of both source and destination
//  using the Ragwitz criteria:
//  a. Auto-embedding method
teCalc.setProperty(TransferEntropyCalculatorKraskov.PROP_AUTO_EMBED_METHOD,
        TransferEntropyCalculatorKraskov.AUTO_EMBED_METHOD_RAGWITZ);
//  b. Search range for embedding dimension (k) and delay (tau)
teCalc.setProperty(TransferEntropyCalculatorKraskov.PROP_K_SEARCH_MAX, "6");
teCalc.setProperty(TransferEntropyCalculatorKraskov.PROP_TAU_SEARCH_MAX, "6");
// Since we're auto-embedding, no need to supply k, l, k_tau, l_tau here:
teCalc.initialise();
// Compute TE from breath (column 1) to heart (column 0)
teCalc.setObservations(MatrixUtils.selectColumn(data, 1), MatrixUtils.selectColumn(d
double teBreathToHeart = teCalc.computeAverageLocalOfObservations();
// Check the auto-selected parameters and print out the result:
int optimisedK = Integer.parseInt(teCalc.getProperty(TransferEntropyCalculatorKrasko
int optimisedKTau = Integer.parseInt(teCalc.getProperty(TransferEntropyCalculatorKra
int optimisedL = Integer.parseInt(teCalc.getProperty(TransferEntropyCalculatorKrasko
```

```
int optimisedLTau = Integer.parseInt(teCalc.getProperty(TransferEntropyCalculatorKra
System.out.printf("TE(breath->heart) was %.3f nats for (heart embedding:) k=%d," +
        "k_tau=%d, (breath embedding:) l=%d,l_tau=%d optimised via Ragwitz criteria\
        teBreathToHeart, optimisedK, optimisedKTau, optimisedL, optimisedLTau);

// Next, embed the destination only using the Ragwitz criteria:
teCalc.setProperty(TransferEntropyCalculatorKraskov.PROP_AUTO_EMBED_METHOD,
        TransferEntropyCalculatorKraskov.AUTO_EMBED_METHOD_RAGWITZ_DEST_ONLY);
teCalc.setProperty(TransferEntropyCalculatorKraskov.PROP_K_SEARCH_MAX, "6");
teCalc.setProperty(TransferEntropyCalculatorKraskov.PROP_TAU_SEARCH_MAX, "6");
// Since we're only auto-embedding the destination, we supply
//  source embedding here (to overwrite the auto embeddings from above):
teCalc.setProperty(TransferEntropyCalculatorKraskov.L_PROP_NAME, "1");
teCalc.setProperty(TransferEntropyCalculatorKraskov.L_TAU_PROP_NAME, "1");
// Since we're auto-embedding, no need to supply k and k_tau here:
teCalc.initialise();
// Compute TE from breath (column 1) to heart (column 0)
teCalc.setObservations(MatrixUtils.selectColumn(data, 1), MatrixUtils.selectColumn(d
double teBreathToHeartDestEmbedding = teCalc.computeAverageLocalOfObservations();
// Check the auto-selected parameters and print out the result:
optimisedK = Integer.parseInt(teCalc.getProperty(TransferEntropyCalculatorKraskov.K_
optimisedKTau = Integer.parseInt(teCalc.getProperty(TransferEntropyCalculatorKraskov
System.out.printf("TE(breath->heart) was %.3f nats for (heart embedding:) k=%d," +
        "k_tau=%d, optimised via Ragwitz criteria, plus (breath embedding:) l=1,l_ta
        teBreathToHeartDestEmbedding, optimisedK, optimisedKTau);
```

JIDT -- Java Information Dynamics Toolkit -- Joseph Lizier *et al.*