.NET Statistical Computation with NMath Stats

CenterSpace™
Software

Introduction

NMath Stats is part of CenterSpace Software's **NMath**™ product suite, which provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. **NMath Stats** provides functions and data structures for statistical computation, including descriptive statistics, probability distributions, combinatorial functions, multiple linear regression, hypothesis testing, and analysis of variance.

Fully compliant with the Microsoft Common Language Specification (CLS), all **NMath Stats** routines are callable from any .NET language, including C#, Visual Basic.NET, and F#.

NOTE—Code samples in this document are shown in C#. Complete NMath Stats code examples in C#, Visual Basic.NET, and F# are available on the CenterSpace website:

`http://www.centerspace.net/examples/NMath/Stats/`

Features

The features of **NMath Stats** include:

- A data frame class for holding data of various types (numeric, string, boolean, datetime, and generic), with methods for appending, inserting, removing, sorting, and permuting rows and columns.
- Functions for computing descriptive statistics, such as mean, variance, standard deviation, percentile, median, quartiles, geometric mean, harmonic mean, RMS, kurtosis, skewness, and many more.
- Probability density function (PDF), cumulative distribution function (CDF), inverse CDF, and random variable moments for a variety of probability distributions, including normal (Gaussian), Poisson, chi-square, gamma, beta, Student's *t*, *F*, binomial, and negative binomial.
- Multiple linear regression and logistic regression.
- Basic hypothesis tests, such as z-test, t-test, F-test, and Pearson's chi-square test, with calculation of p-values, critical values, and confidence intervals.

- One-way and two-way analysis of variance (ANOVA) and analysis of variance with repeated measures (RANOVA).
- Non-parametric tests, such as the Kolmogorov-Smirnov test and Kruskal-Wallis rank sum test.
- Multivariate statistical analyses, including principal component analysis, factor analysis, hierarchical cluster analysis, and *k*-means cluster analysis.
- Nonnegative matrix factorization (NMF), and data clustering using NMF.
- Partial least squares (PLS).
- Statistical process control.
- Visualization using the Microsoft Chart Controls for .NET.

Design

NMath Stats is built on **NMath**, the foundational library in the **NMath** product suite. **NMath** includes general vector and matrix classes, complex number classes, random number generators, and numerical integration methods. For more information on **NMath**, see *.NET Numerical Applications with NMath* (Technical Report #1, CenterSpace Software).

Data Frames

The statistical functions in **NMath Stats** support the **NMath** types **DoubleVector** and **DoubleMatrix**, as well as simple arrays of doubles. In many cases, these types are sufficient for storing and manipulating your statistical data. However, they suffer from two limitations: they can only store numeric data, and they have limited support for adding, inserting, removing, and reordering data. Because the underlying data is an array of doubles, data must be copied to new storage every time manipulation operations such as these are performed.

For these reasons, **NMath Stats** provides the **DataFrame** class which represents a two-dimensional data object consisting of a list of columns of the same length. Columns are themselves lists of different types of data: numeric, string, boolean, generic, and so on.

A **DataFrame** can be viewed as a kind of virtual database table. Columns can be accessed by numeric index (`0...n-1`) or by a string name supplied at construction time. Rows can be accessed by numeric index (`0...n-1`) or by a key object.

Data frames can be constructed in a variety of ways. The default constructor creates an empty data frame with no rows or columns. Columns and rows can then be added to the new data frame. For example:

```
var df = new DataFrame();

// Add some columns
df.AddColumn( new DFStringColumn( "Sex" ) );
df.AddColumn( new DFStringColumn( "AgeGroup" ) );
df.AddColumn( new DFNumericColumn( "Weight" ) );

// Add some rows
df.AddRow( "John Smith", "M", "Child", 45 );
df.AddRow( "Ruth Barnes", "F", "Senior", 115 );
df.AddRow( "Jane Jones", "F", "Adult", 115 );
df.AddRow( "Timmy Toddler", "M", "Child", 42 );
df.AddRow( "Betsy Young", "F", "Adult", 130 );
df.AddRow( "Arthur Smith", "M", "Senior", 142 );
df.AddRow( "Lucy Doe", "F", "Child", 30 );
```

This data frame contains three columns: column `0`, named `Sex`, contains string data; column `1`, named `AgeGroup`, also contains string data; column `2`, named `Weight`, contains numeric data. There are seven rows of data in this data frame, and the subjects' names are used as row keys.

Methods are provided for appending, inserting, removing, sorting, and permuting rows and columns in a data frame. This code manipulates a data frame:

```
// switch last two columns
df.PermuteColumns( 0, 2, 1 );

// sort rows primarily by AgeGroup in ascending order,
// and secondarily by Sex in descending order
int[] colIndices = { 2, 0 };
SortingType[] sortingTypes = { SortingType.Ascending,
                               SortingType.Descending };
df.SortRows( colIndices, sortingTypes );

// delete a row by key
df.RemoveRow( "Lucy Doe" );

// export to a ADO.NET DataTable
DataTable dt = df.ToDataTable();
```

Because the underlying data is in a list, elements can be added, removed, and reordered without having to copy all of the data to new storage.

Subsets

In addition to accessors for individual elements, columns, or rows in a data frame, class **DataFrame** provides a large number of indexers and member functions for accessing sub-frames containing any arbitrary subset of rows, columns, or both.

Such indexers and methods accept the **NMath** types **Slice** and **Range** to indicate sets of row or column indices with constant spacing. In addition, **NMath Stats** introduces a new class called **Subset**. Like a **Slice** or **Range**, a **Subset** represents a collection of indices that can be used to view a subset of data from another data structure. Unlike a **Slice** or **Range**, however, a **Subset** need not be continuous, or even ordered. It is simply an arbitrary collection of indices.

For example, this code gets a new data frame containing columns `3-8` in reverse order, and all rows where the value in column `0` in greater than the value in column `1`:

```
var colRange = new Range( 8, 3, -1 );

var bArray = new bArray[ df.Rows ];
for ( int i = 0; i < df.Rows; i++ )
{
  bArray[i] = ( df[0][i] > df[1][i] );
}
var rowSubset = new Subset( bArray );

DataFrame df2 = df[ rowSubset, colRange ];
```

This code utilizes a very useful **Subset** constructor which takes an array of boolean values and constructs a **Subset** containing the indices of all `true` elements in the array.

Factors

The **Factor** class represents a categorical vector in which all elements are drawn from a finite number of factor levels. Thus, a **Factor** contains two parts: an object array of factor levels, and an integer array of categorical data, of which each element is an index into the array of levels. For example, this string data:

```
"A", "A", "C", "B", "A", "C", "B"
```

could be presented as a **Factor** with the following levels and categorical data:

```
object[] levels = { "A", "B", "C" };
int[] data = { 0, 0, 2, 1, 0, 2, 1 };
```

Factors are usually constructed from a data frame column using the `GetFactor()` method, which creates a **Factor** with levels for the sorted, unique values in the column.

The principal use of factors is in conjunction with the `GetGroupings()` methods on **Subset**. One overload of this method accepts a single **Factor** and returns an array of subsets containing the indices for each level of the given factor. Another overload accepts two **Factor** objects and returns a two-dimensional jagged array of subsets containing the indices for each combination of levels in the two factors.

For example, this code constructs factors from the values in the `Sex` and `AgeGroup` columns of a data frame, then uses these factors in conjunction with the `GetGroupings()` methods on **Subset** to create subsets representing the groups for each level of the factors, as well as all combinations of the factors:

```
Factor sex = df.GetFactor( "Sex" );
Factor age = df.GetFactor( "AgeGroup" );

Subset[] sexGroups = Subset.GetGroupings( sex );
Subset[] ageGroups = Subset.GetGroupings( age );
Subset[,] cellGroups = Subset.GetGroupings( sex, age );
```

These subsets can then be used to operate on the relevant portions of the data frame, as described above.

Descriptive Statistics

Class **StatsFunctions** provides a wide variety of static functions for computing descriptive statistics, such as mean, variance, standard deviation, percentile, median, quartiles, geometric mean, harmonic mean, RMS, kurtosis, skewness, and many more. Method overloads accept data as an array of doubles, as a **DoubleVector**, or as a numeric column in a **DataFrame**. For example:

```
double[] dblArray = { 1.12, -2.0, 3.88, 1.2, 15.345 };
double mean1 = StatsFunctions.Mean( dblArray );

var v = new DoubleVector( "1.12 -2.0 3.88 1.2 15.345"  );
double mean2 = StatsFunctions.Mean( v );

var df = new DataFrame();
df.AddColumn(
  new DFNumericColumn( "myData", 1.12, -2.0, 3.88, 1.2, 15.345 ) );
double mean3 = StatsFunctions.Mean( df[ "myData" ] );
```

Most functions in class **StatsFunctions** are accompanied by a paired function which ignores values that are Not-a-Number (NaN). For example, there are `Mean()` and `NaNMean()` functions, `Variance()` and `NaNVariance()` functions, and

so forth. Unless a function is explicitly designed to handle missing values, it may return NaN or have unexpected results if values are missing.

Probability Distributions

NMath Stats provides classes for computing the probability density function (PDF), the cumulative distribution function (CDF), the inverse cumulative distribution function, and random variable moments for a variety of probability distributions, including normal (Gaussian), Poisson, chi-square (χ^2), gamma, beta, Student's *t*, *F*, binomial, and negative binomial. For example, this code constructs an *F* distribution object with degrees of freedom `11,19`, then queries it:

```
int df1 = 11;
int df2 = 19;
var dist = new FDistribution( df1, df2 );

double pdf = dist.PDF( 1.45 );
double cdf = dist.CDF( 1.45 );
double invCdf = dist.InverseCDF( .95 );
double mean = dist.Mean;
double var = dist.Variance;
double kurt = dist.Kurtosis;
double skew = dist.Skewness;
```

Hypothesis Tests

NMath Stats provides classes for many common hypothesis tests, such as the z-test, t-test, F-test, and Kolmogorov-Smirnov test, with calculation of p-values, critical values, and confidence intervals.

For example, class **OneSampleZTest** determines whether a sample from a normal distribution with known standard deviation could have a given mean. If the sample data is vectors `data`, this code constructs a hypothesis test object:

```
double mu0 = 100;
double sigma = 15;
var test = new OneSampleZTest( data, mu0, sigma );
```

By default, a **OneSampleZTest** object performs a two-sided hypothesis test ($H_1: \mu \neq \mu_0$) with $\alpha = 0.01$. Non-default test parameters can be specified at the time of construction using constructor overloads, or after construction using the provided `Type` and `Alpha` properties:

```
test.Type = HypothesisType.Right;
test.Alpha = 0.05;
```

All hypothesis test classes provide a `ToString()` method that returns a formatted string representation of the test results:

```
One Sample Z Test
-----------------

Sample mean = 112.8
Sample size = 9
Population mean = 100
Population standard deviation = 15
Computed Z statistic: 2.56

Hypothesis type: one-sided to the right
Null hypothesis: sample mean = population mean
Alt hypothesis: sample mean > population mean
P-value: 0.00523360816355578
REJECT the null hypothesis for alpha = 0.05
0.95 confidence interval: 104.575731865243 Infinity
```

Properties are also provided for accessing individual elements in the test results.

Linear Regression

Class **LinearRegression** computes a multiple linear regression from an input matrix of independent variable values (the *predictor matrix* or *regression matrix*) and a vector of dependent variable values (the *observation vector*). For example:

```
var predictors =
  new DoubleMatrix( " 8x4 [ 1 1450 .50 70
                            1 1600 .50 70
                            1 1450 .70 70
                            1 1600 .70 70
                            1 1450 .50 120
                            1 1600 .50 120
                            1 1450 .70 120
                            1 1600 .70 120 ]" );
var obs = new DoubleVector( "[ 67 79 61 75 59 90 52 87 ]" );
var lr = new LinearRegression( A, obs );
```

By default, model parameter values are computed by the *method of least squares* using a QR factorization, but you may elect to use a complete orthogonal factorization or singular value decomposition instead.

The *y*-intercept is the first element of the parameter array returned by the regression, and the slope is the second:

```
Console.WriteLine( "y-intercept = {0}", regression.Parameters[0] );
Console.WriteLine( "Slope = {0}", regression.Parameters[1] );
```

You can also use a linear regression object to generate predictions:

```
var predictors = new DoubleVector( 150.0, 33.5, 0.66, 80.0 );

double predicted = lr.PredictedObservation( predictors );
```

NMath Stats also provides the **LinearRegressionParameter** class for testing statistical hypotheses about individual parameters in a **LinearRegression**. This code tests whether the fifth parameter in a model is significantly different than zero:

```
var param = new LinearRegressionParameter( lr, 4 );
double tstat = param.TStatistic( 0.0 );
double pValue = param.TStatisticPValue( 0.0 );
double criticalValue = param.TStatisticCriticalValue( 0.05 );
Interval confidenceInterval = param.ConfidenceInterval( 0.05 );
```

Class **LinearRegressionAnova** tests the overall model significance for linear regressions:

```
var lrAnova = new LinearRegressionAnova( lr );
double sse = lrAnova.ResidualSumOfSquares;
double r2 = lrAnova.RSquared;
double fstat = lrAnova.FStatistic;
double fstatPval = lrAnova.FStatisticPValue;
```

Analysis of Variance

NMath Stats provides classes for both *one-way* (one-factor) and *two-way* (two-factor) ANOVAs. One-way ANOVA is supported for both balanced and unbalanced designs, and with or without repeated measures (RANOVA). Two-way ANOVA is supported for balanced designs only, with or without repeated measures.

For example, this code constructs a two-way ANOVA by grouping the numeric data in column `3` of **DataFrame** `df` by factors constructed from columns `0` and `4`:

```
var anova = new TwoWayAnova( df, 0, 4, 3 );
```

Once you've constructed an ANOVA object, you can display the complete ANOVA table:

```
Console.WriteLine( anova );
```

For example:

```
Source  Deg of Freedom  SumOfSq    Mean Square  F         P
FactorA       1         1782.0450  1782.0450    14.2121   0.0008
FactorB       1         2838.8113  2838.8113    22.6399   0.0001
```

```
Interaction  1          108.0450   108.0450     0.8617    0.3612
Error        28         3510.9075  125.3896     .         .
Total        31         8239.8088  .
```

Properties are provided for accessing individual elements in the ANOVA table.

NMath Stats solves the two-way ANOVA problem using multiple linear regression. If all you wish to know is the information in the standard ANOVA table, you can safely ignore the regression details, but properties and member functions are provided for retrieving information about the underlying regression parameters.

Multivariate Statistics

NMath Stats provides classes for dimension reduction using *principal component analysis* (PCA) and *factor analysis*, and case reduction using *hierarchical cluster analysis* and *k-means cluster analysis*. Principal component analysis finds a smaller set of synthetic variables that capture the variance in an original data set. The first principal component accounts for as much of the variability in the data as possible, and each succeeding orthogonal component accounts for as much of the remaining variability as possible. A **FloatPCA** or **DoublePCA** instance is constructed from a matrix or a dataframe containing numeric data. Each column represents a variable, and each row represents an observation. The data may optionally be zero-centered and scaled to have unit variance.

```
bool center = true;
bool scale = true;
var pca = new DoublePCA( data, center, scale );
Console.WriteLine( "Loading Martrix = " + pca.Loadings );
Console.WriteLine( "Variance Proportions = " +
                   pca.VarianceProportions );
Console.WriteLine( "Cumulative Variance Proportions = " +
                   pca.CumulativeVarianceProportions );
Console.WriteLine( "Scores = " + pca.Scores );
```

Hierarchical cluster analysis detects natural groupings in data. Each object is initially assigned to its own singleton cluster. The analysis then proceeds iteratively, at each stage joining the two most similar clusters into a new cluster, continuing until there is one overall cluster.

During clustering, the distance between individual objects is computed using a distance function delegate. Delegates are provided as static variables on class **Distance** for euclidean, squared euclidean, city-block (Manhattan), maximum (Chebychev), and power distance functions. You can also create your own distance function delegate. The distances between clusters of objects are computed using a linkage function delegate. Delegates are provided as static variables on class **Linkage** for single, complete, unweighted average, weighted average, centroid, median, and Ward's linkage functions. Again, you can also create your own

linkage function delegate. For example, this code clusters `8` random vectors of length `3`:

```
var data = new DoubleMatrix( 8, 3, new RandGenUniform() ); var ca =
new ClusterAnalysis( data, Distance.SquaredEuclideanFunction,
  Linkage.CompleteFunction );
```

Property `Distances` gets the vector of distances between all possible object pair; `Linkages` gets the complete hierarchical linkage tree. The `CutTree()` method constructs a set of clusters by cutting the hierarchical linkage tree either at the specified height, or into the specified number of clusters.

```
Console.WriteLine( ca.Distances );
Console.WriteLine( ca.Linkages );

// cut linkage tree to form 3 clusters Console.WriteLine(
ca.CutTree( 3 ) );

// cut linkage tree at height of 0.75 Console.WriteLine(
ca.CutTree( 0.75 ) );
```

Nonnegative Matrix Factorization

Nonnegative matrix factorization (NMF) factors a matrix V into two matrices, W and H. NMF differs from many other factorizations by enforcing the constraint that the factors W and H must be non-negative—that is, all elements must be equal to or greater than zero.

If a set of n-dimensional m data vectors are placed in an n x m matrix V, then NMF can be used to approximately factor V into an n x r matrix W and an r x m matrix H. Usually r is chosen to be much smaller than either m or n, so that W and H are smaller than the original matrix V. Thus, each column v of V is approximated by a linear combination of the columns of W, with the coefficients being the corresponding column of H, $v \approx Wh$. This extracts underlying features of the data as basis vectors in W, which can then be used for identification and classification. By not allowing negative entries in W and H, NMF enables a non-subtractive combination of the parts to form a whole.

NMath Stats provides classes for **NMFact** for basic NMF, and **NMFClustering** for data clustering using NMF. For example, the following code clusters data in a **DoubleMatrix** using NMF, and prints the results:

```
DoubleMatrix data = ...   // data to be factored
int k = ...               // number of columns in W

var nmfClustering = new NMFClustering<NMFDivergenceUpdate>();

nmfClustering.Factor( data, k );
```

```
ClusterSet cs = nmfClustering.ClusterSet;

// Print out the cluster each column belongs to
for ( int i = 0; i < cs.N; i++ ) {
  Console.WriteLine( "Column {0} belongs to cluster {1}",
    i, cs[i] );
}

// Print out the the members of each cluster
for ( int i = 0; i < cs.NumberOfClusters; i++ ) {
  int[] members = cs.Cluster( i );
  Console.Write( "Cluster number {0} contains: ", i );
  for ( int j = 0; j < members.Length; j++ ) {
    Console.Write( "{0} ", j );
  }
  Console.WriteLine();
}
```

Since NMF uses random starting values for W and H, and the factorization is not unique, you can get different clusterings for the columns of V on different runs. A *consensus matrix* is a way to average multiple clusterings, to produce a probability estimate that any pair of columns will be clustered together. **NMath Stats** provides class **NMFConsensusMatrix** for compute a consensus matrix using NMF.

Partial Least Squares

Partial Least Squares (PLS) is a technique that generalizes and combines features from principal component analysis and multiple linear regression. It is particularly useful when you need to predict a set of response (dependent) variables from a large set of predictor (independent variables). **NMath Stats** supports both the Nonlinear Iterative PArtial Least Squares (NIPALS) and Straightforward IMplementation of Partial Least Squares (SIMPLS) algorithms.

NMath Stats provides two classes for performing PLS regression:

- **PLS1** is used when the responses, Y, in the model $Y=XB+E$ consist of a single variable. In this case Y is a vector containing the n response values.
- **PLS2** is used when the responses are multivariate. In this case Y is a matrix composed of n rows with each row containing the m response variable values.

Computing a PLS regression is accomplished by simply constructing a **PLS1** or **PLS2** instance. The basic parameters are:

- the matrix of predictor variables values
- the response variable values (a vector for **PLS1** and a matrix for **PLS2**)

- an integer specifying the number of factors or components

For example:

```
DoubleMatrix A = ...
DoubleVector y = = ...
int numComponents = 3;

var pls = new PLS1( A, y, numComponents );
```

NMath Stats also provides the classes **PLS1Anova** and **PLS2Anova** for performing a classic ANOVA for **PLS1** and **PLS2** regression models. These classes calculate the sum of squares total, sum of squares residual, mean square error for prediction, and the coefficient of determination.

Visualization

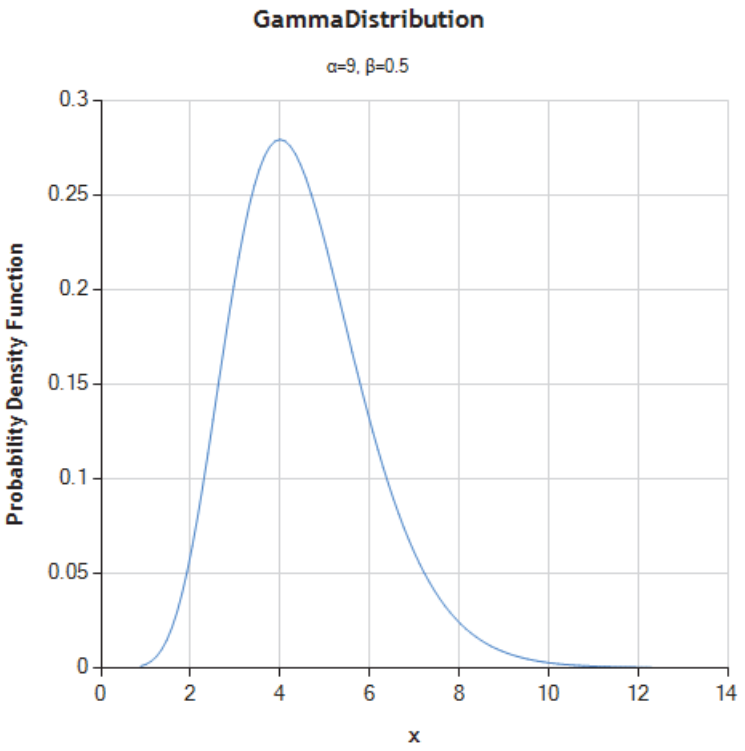
NMath Stats can be easily combined with the free Microsoft Chart Controls for .NET to create a complete data analysis and visualization solution. The Microsoft Chart Controls for .NET are available as a separate download for .NET 3.5. Beginning in .NET 4.0, the Chart controls are part of the .NET Framework.

NMath Stats provides convenience methods for plotting **NMath Stats** types using the Microsoft Chart Controls. For example, this code plots the probability density function (PDF) of the specified gamma distribution:

```
double alpha = 9.0;
double beta = 0.5;
var gamma = new GammaDistribution( alpha, beta );

NMathStatsChart.Show( gamma,
  NMathStatsChart.DistributionFunction.PDF );
```

Figure 1 – Gamma distribution PDF



For more information, see CenterSpace whitepaper "NMath Stats Visualization Using the Microsoft Chart Controls."

Conclusions

NMath Stats is a powerful, easy-to-use component library for data manipulation and statistical analysis on the .NET platform.

Fully compliant with the Microsoft Common Language Specification, all **NMath Stats** routines are callable from any .NET language, including C#, Visual Basic.NET, and Managed C++.

.NET Statistical Computation with NMath Stats

The correct bibliographic reference for this document is:
.NET Statistical Computation with NMath Stats, CenterSpace Software, Corvallis, OR.

Printed in the United States.
Printing Date: March, 2016

CenterSpace Software

Address:	622 NW 32nd St., Corvallis, OR 97330 USA
Phone:	(541) 896-1301
Web:	http://www.centerspace.net
Technical Support:	support@centerspace.net