

Technical
White Paper

Using the Powerful Classes for
Business Intelligence in
Analytics.h++



www.roguewave.com

Using the Powerful Classes for Business Intelligence in **Analytics.h++**

Table of contents

Introducing Analytics.h++ 1

- Advantages of Analytics.h++ 1
- Data manipulation tools 2
- High level mathematical expressions 2
- Vector and matrix “views” to slice and dice your data 2
- Example code 3

Data preprocessing tool: singular value decomposition 3

- Example: predicting spending levels from previous spending habits 4
- Statistical prediction tools 4
- Linear regression 5
- Logistic regression 5

Tools for determining confidence levels 7

- Confidence levels for model parameters 7

Tools for comparing statistical models 8

- Example: eliminating variables confidently 8

Model selection algorithms 9

- Model selection viewed as search 9
- Example code 10

Summary 11

Introducing Analytics.h++

Understanding your data is critical to understanding your business. Timely business decisions are increasingly reliant upon the computer-assisted collection and analysis of key data. To remain competitive in the face of increasingly shorter business cycles, you must be able to analyze a mountain of data to find those pieces of information that will lead to more profitability. This means that applications that simplify access to *business intelligence*—also referred to as *data mining* and *decision support systems*—are essential ingredients within your application portfolio.

Rogue Wave Software offers Analytics.h++, a collection of powerful and intuitive C++ classes that make it simpler to write business intelligence applications. Analytics.h++ provides the analytical tools necessary to give structure, order, and form to your business data. By allowing you to focus on designing and building the business data models in your custom applications, instead of on the underlying data structures and algorithms to analyze them, Analytics.h++ offers real productivity advantages. It includes a full complement of algorithms for statistical data analysis, matrix decomposition and analysis, eigenvalue analysis, and Fourier analysis. This broad collection of tools is useful for a variety of business intelligence applications.

Advantages of Analytics.h++

Analytics.h++ offers many advantages over other commercially available data analysis libraries. Here are a few of the main advantages:

- **Powerful classes for statistical prediction.** In just a few lines of code, it is possible to form statistical predictions using linear and logistic regression, to provide confidence levels for the accuracy of predictions and model parameters, and to automatically search for the best possible predictive model.
- **Fast development of business intelligence applications.** Interfaces to Analytics.h++ classes are intuitive and easy to learn. For example, the addition of two matrices, A and B, and subsequent assignment to a matrix C, is expressed as “C = A + B.” Also, classes give you a high level of abstraction, so instead of investing months of development effort in writing low-level algorithms, you'll write your business logic in just a few days.
- **Compact data storage.** Rather than each matrix or vector holding duplicate copies of a collection of values, data stored in a single memory location can be shared among many matrices and vectors. Each vector or matrix has its own “view” of the data segment, enabling access to a structured subset of another matrix or vector's data.
- **Fast numerical processing.** Optimized numerical operations that use the underlying hardware for maximum processing speed are included for many hardware configurations, including Intel, Hewlett-Packard, and Sun. These operations include matrix-vector multiplication and dot product.

- **Hierarchical class interfaces.** Unlike libraries written in C or FORTRAN, class interfaces are organized in a hierarchy. This makes it easy to find and understand a related group of algorithms, and to ignore algorithms not useful for developing the current application.
- **Easily modifiable class interfaces.** Analytics.h++ was written with extensibility in mind. It allows you to quickly create your own “flavor” of an existing class. If you already have certain algorithms that you know and trust, these can quickly be incorporated into an Analytics.h++ class through class inheritance.
- **Easy migration from FORTRAN and C.** Don’t be concerned if you have already invested significant programming effort in algorithms that you want to use: Analytics.h++ gives you a path to migrate your code to C++. Code can easily be added to the extensible framework, and there is support in Analytics.h++ for matching storage schemes of C++ vector and matrix classes with those expected by routines written in C or FORTRAN.

The remaining sections of this report provide further details and in-depth examples for some of the major tools found in Analytics.h++. This includes tools for data manipulation, data preprocessing, and statistical prediction, as well as tools for determining confidence levels.

Data manipulation tools

The vector and matrix classes included with Analytics.h++ are efficient data manipulation tools. They increase programmer efficiency with a high-level, easily understood interface while allowing highly efficient semantics for storing and accessing data values. These features are discussed separately below.

High level mathematical expressions

Analytics.h++ allows you to express computations using intuitive, high level mathematical notation. This markedly reduces the time it takes to learn the class interfaces and begin writing effective programs. It provides simple algebraic operators (+, -, *, /), along with matrix-vector multiplication and inner product. Mathematical functions, including sqrt, abs, pow, ceil, and floor, are also available.

Vector and matrix “views” to slice and dice your data

Because many numerical algorithms are based upon incrementally updating matrices and vectors, Analytics.h++ allows the creation of a subvector that “views” the slice of another vector or matrix, or a submatrix that “views” the slice of another matrix. (In this context, *slice* means a structured subset of elements from a matrix or vector.) Matrix and vector views are analogous to a pivot table in some spreadsheet programs.

Views are also useful when you want a slice from a multidimensional data model used in OLAP query tools. Using views, you can update all elements in a submatrix or

subvector using a single command. By contrast, other numerical libraries that don't allow views require that you either create extra, temporary structures to perform incremental operations, or explicitly loop over each element in the slice, performing the same operation on each element. The matrix and vector views in `Analytics.h++` do not cause the duplication of data values. They store only the index of the first value in the view, the number of elements in the view, and the *stride*—the distance between values in the original matrix or vector. Using views results in shorter, more intuitive programs that avoid creating unnecessary, temporary storage.

Example code

The following simple example shows how intuitive the code for vector and matrix manipulation can be. The following code creates a view of data and decrements the elements with even-valued indexes by one.

```
// Define a vector and a matrix.
RWMathVec<double> vector;
RWGenMat<double> matrix;

// Read in the vector, then the matrix.
cin >> vector >> matrix;

// Compute a high-level expression.
RWMathVec<double> answer = log(vector) + product(matrix, vector);

// Create a vector view that references the even elements
// of "answer". The three parameters indicate the starting
// index, number of elements, and "stride" in the view.

RWSlice evenvals(1, answer.length() / 2, 2);

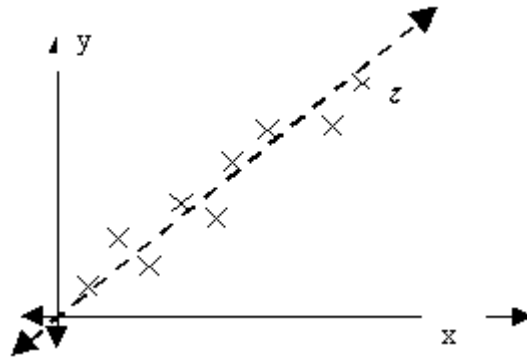
// Decrement the elements with even-valued indexes by one.
answer(evenvals) -= 1.0;

// Print out the results.
cout << "answer = " << answer << "\n";
```

Data preprocessing tool: singular value decomposition

Certain numerical algorithms improve the predictive power of statistical models by “cleaning up” the data. An important algorithm included with `Analytics.h++` called Singular Value Decomposition (SVD) solves such problems as multicollinearity, near-singularity, and “noise” in data caused by measurement error.

SVD creates a new set of coordinate axes for a collection of data points. The axes are oriented along the “directions of maximum variation” in the data. In Figure 1, notice that the data points, originally expressed in x, y coordinates, lie roughly in a line. The SVD creates a new coordinate space (marked with dashed lines) where the new z axis is oriented along the line suggested by the data. Now a point’s location can be described accurately by specifying a value for z , rather than by specifying values for an x, y pair.



*Figure 1. Example of an SVD transformation.
Data point locations are represented with x marks.
The SVD's created coordinate axis is represented with a dashed line.*

Although there are no intuitive, associated labels for the created axes, which limits interpretive ability, predictive ability is increased. When using the resulting data coordinates in regression, the estimated regression parameters tend to be better conditioned, and the predictions tend to be more accurate.

Example: predicting spending levels from previous spending habits

Suppose that a credit analyst is interested in creating a statistical model for predicting a person's future spending behavior from historical information. This historical information consists of the last three monthly savings balances and the current level of credit card debt.

The analyst hypothesizes that the three monthly savings balance variables are highly correlated and decides to perform an SVD using *Analytics.h++*. As expected, the transformation reduces the number of data dimensions from four (three savings balance variables and one credit card debt variable) to three. Instead of the data filling a four-dimensional space, the points fall roughly on a three-dimensional linear subspace. Using the resulting three variables, the analyst finds that statistical predictions of spending behavior are more accurate than when the original four variables were used.

Statistical prediction tools

Analytics.h++ gives you two powerful classes for statistical prediction, the backbone of your business intelligence application. These tools, *RWLinearRegression* and *RWLogisticRegression*, are good for predicting the outcome of some measurable event, such as whether a sale will be made, or the future price of a commodity. They can also be used to establish the degree to which two variables, such as price and level of satisfaction, are related.

Linear regression

Linear regression is useful when the intended prediction is a continuously varying amount—such as number of hours or price per unit—and is commonly used in many statistics applications, including price forecasting, trend analysis and cyclical variation. A linear regression model interpolates the data it is given with a linear function and uses this function to produce its best guess for future values. By examining the parameters estimated by the linear regression, it is also possible to determine which variables are most helpful in a prediction.

Example: analysis of sales factors

Suppose a market research analyst is interested in determining which of ten factors most influences the sales of a product. The ten factors include such measurable amounts as time of year and price per unit. Data on sales levels and all ten factors have been collected weekly for the past two years, resulting in a 104 x 10 matrix of factor values and a length-104 vector of sales level values.

Using `Analytics.h++`, the analyst forms a linear regression model that predicts sales levels from the ten factors. Examination of the model's estimated parameters shows values ranging in magnitude from 0.04 to 1.54. The factor associated with the parameter of 1.54, the one with the highest magnitude, apparently has the highest influence on sales.

The following lines of code show how the analyst writes the application that performs the above steps.

```
// Create space for the data.
RWGenMat<double> factorMatrix;
RWMathVec<double> salesVector;

// Read in the data values.
cin << factorMatrix << salesVector;

// Construct the linear regression model.
RWLinearRegression lr(factorMatrix, salesVector);

// Find and output the value of the model parameter
// with the largest absolute value.
RWMathVec<double> sortedParams = sort(abs(lr.parameters()));
cout >> "largest magnitude: " >> sortedParams(lr.numPredictors());
```

Logistic regression

Logistic regression is used when the event to be predicted is either true or false, such as whether a future sale will be successful. The model produces a probability, between zero and one, of a successful outcome. It shares many of the same uses as linear regression: the only important difference between the two models is that logistic regression is appropriate for predicting Boolean-valued outcomes, and linear regression is appropriate for predicting continuous-valued outcomes.

Example: predicting the probability of a successful sale

Let's assume that a company selling long distance access has a large list of potential long distance customers, but a sales force that can handle calling only ten percent of those potential customers. The analyst has collected data on thirty thousand previous calls, some successful and some unsuccessful in selling long distance. The collected data for each past and potential customer includes age, income level, current long distance rates, and number of switches to a new long distance company in the past year.

The analyst decides to perform a logistic regression model that will find the ten percent of potential customers that are most likely to buy long distance service. The following code shows how the analyst would accomplish this using `Analytics.h++`.

```
// The model has four variables from which the
// probability of selling long distance is predicted:
// 1: Age
// 2: Income Level
// 3: Current Long Distance Rate
// 4: Number of Switches In Last Twelve Months

// Specify the sizes of matrices and vectors.
const size_t numVariables = 4;
const size_t numPastCalls = 30000;
const size_t numPotentialBuyers = 10000;

// Create space for the data based on previous calls.
RWGenMat<double> oldCallData(numPastCalls, numVariables,
                             rwUninitialized);
RWMathVec<double> oldOutcomes(numPastCalls, rwUninitialized);

// Create space for the potential customer data.
RWGenMat<double> potentialBuyers(numPotentialBuyers, numVariables,
                                 rwUninitialized);

// Read in the data values.
cin << oldCallData << oldOutcomes << potentialBuyers;

// Construct the logistic regression model.
RWLogisticRegression lr( oldCallData, oldOutcomes );

// Find and output the probabilities of potential
// customers buying long distance according to the model.
RWMathVec<double> computedProbs =

lr.predictedProbSuccess(potentialBuyers);
cout >> "computed probabilities: " >> computedProbs;
```

Tools for determining confidence levels

While some businesses are successful with the predictions produced by regression models, there is an extra level of information readily available in `Analytics.h++`. Not only do the regression models produce predictions, but they can also produce confidence levels for the accuracy of model parameters and predictions. This extra information helps a business decide the degree to which they can trust a model's predictions.

Confidence levels for model parameters

Recall the example where a market research analyst finds the most influential factor in selling a product. The most influential factor was the one associated with the largest model parameter value of 1.54. Now suppose that the next-largest parameter has a value of 1.53. The analyst is forced to ask whether, for all practical purposes, the two factors are identically influential. Perhaps if the data had been slightly different, the second factor would have appeared even slightly more influential than the first.

As part of a linear and logistic regression class interface, `Analytics.h++` includes tools for computing confidence levels for estimated parameter values. For example, the analyst can ask whether there is a 95% certainty that the largest parameter is actually the most influential.

Example: confidence in largest sales factor

The following code shows how the market research analyst would discover whether there is strong evidence for one factor being more influential than the rest. The example assumes that the factors have been re-ordered in a new matrix, such that the first column of data is associated with the most influential factor, and the second column is associated with the second most influential factor, and so on.

```
// Create space for the data.
RWGenMat<double> orderedFactorMatrix;
RWMathVec<double> salesVector;

// Read in the data values.
cin << orderedFactorMatrix << salesVector;

// Construct the linear regression model.
RWLinearRegression lr(orderedFactorMatrix, salesVector);

// Form a list of estimated probability distributions
// for the model parameters.
RWTValSlist< RWLinearRegressionParam > probDistList =
lr.parameterEstimates();

// Find and print the degree to which one can know
// that the first is larger than the second.
double testValue = probDistList[1].value();
double confLevel = probDistList[0].tStatisticPvalue(testValue);
cout >> "Confidence in Largest Factor: " >> confLevel;
```

Tools for comparing statistical models

Statistical prediction models should be constantly refined. A statistical analyst should constantly be asking, “Is there some improvement that I can make to the current model?” To assist this process, Analytics.h++ also includes tests for comparing two linear or logistic regression models. The tests can be used to identify the best model from an entire list of possibilities.

Example: eliminating variables confidently

Let us continue with the example of the long distance company we discussed previously. The company’s logistic regression model has consistently produced a parameter value estimate that is close to zero for the age variable. The analyst wonders if this variable is helping predictions at all: perhaps the model would be improved by eliminating this variable altogether. Note that the company could just as easily determine whether adding a replacement variable for age would help even further. The following code shows how the company could test whether a regression model would be improved by eliminating a particular variable.

```
// Repeated code from the previous example:
// construct the logistic regression model.
RWLogisticRegression lr(oldCallData, oldOutcomes);

// Use the Hosmer-Lemeshow statistic for logistic
// regression models to assign a statistical value to
// the current model's predictive accuracy.
RWLogisticFitAnalysis currentFit(lr);
double currentAccuracy = currentFit.HLStatisticPValue();

// Remove the "Age" variable (the first predictor
// variable) from the logistic regression model. The
// code line below says "Remove one variable starting
// at index zero".
Lr.removePredictors(1, 0);

RWLogisticFitAnalysis newFit(lr);
double newAccuracy = newFit.HLStatisticPValue();

// Report whether eliminating the "Age" variable improves
// model accuracy or not.
if (newAccuracy > oldAccuracy) {
    cout >> "Removing the variable improved prediction accuracy.";
}
else {
    cout >> "Removing the variable hurt prediction accuracy.";
```

Model selection algorithms

Analytics.h++ includes a number of high-level algorithms that can be used for model selection. These algorithms are extremely useful for applications where a linear or logistic regression model has a large number of potential variables, of which only a fraction may be useful for prediction. The algorithms conduct model selection as a search for the ideal set of predictor variables. Analytics.h++ provides classes that encapsulate four kinds of search algorithms: *forward*, *backward*, *stepwise*, and *exhaustive selection*.

Model selection viewed as search

When model selection is viewed as a search, the search space consists of possible subsets of predictor variables. An evaluation criterion assigns each subset a numerical value, and the goal of the search is to find the subset with the highest numerical value. The only difference among the techniques involves the choice for a starting subset and the specification of neighboring subsets in the search space.

As an example, Figure 2 illustrates one model selection technique, forward selection, using a connected graph. In this particular example, there are six possible predictor variables: $\{X_1, \dots, X_6\}$. Nodes in the graph indicate predictor variable subsets; each node is labeled with the subset, a bit representation of the subset, and the subset's evaluation criterion. The current node in the search is shaded, and its neighboring nodes are not shaded. Graph edges indicate which subsets are considered neighbors in the search; edges are labeled with the action that changes the current subset into its neighbor. Search continues along the path in the graph that leads to the greatest evaluation criterion, and stops when the evaluation criterion can no longer improve. In the graph in Figure 4, forward selection adds the predictor variable X_5 because the resulting predictor variable subset has the highest evaluation criterion, with a value of 6.04. Forward selection then makes this subset the current one and continues by evaluating the new subset's neighbors. The new neighbors are not shown in the figure.

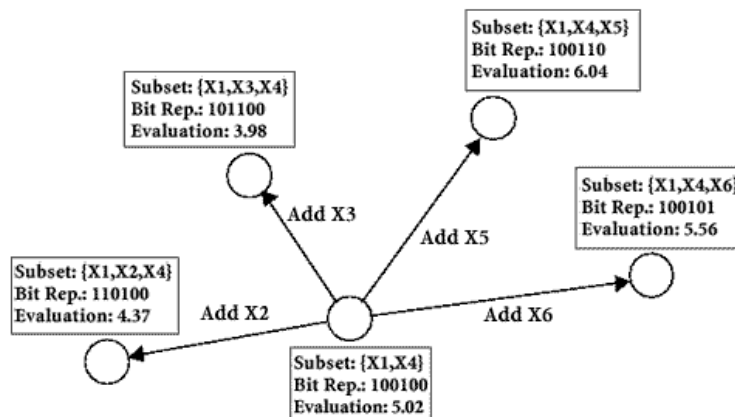


Figure 2. Graphical representation of forward selection as search

Example code

The following example shows how to perform forward selection using the `Analytics.h++` class `RWLinRegModelSelector<F>` on a linear regression problem. We begin with a double-precision predictor matrix called `predictorData` and a double-precision observation vector called `observationData`. The next three lines create a linear regression model and a model selector object set to use forward selection with the F statistic as the subset evaluation criterion.

```
// Save some typing
typedef RWLinRegModelSelector<RWLinRegressFStatistic>
    FStatModelSelector;
RWLinearRegression lr(predictorData, observationData);
FStatModelSelector selector(lr, rwForwardSelection);
```

The next few lines examine the results of forward selection and print out key diagnostics. These diagnostics include a bit vector showing which predictor variables were selected, the parameter values associated with the selected predictor variables, and the evaluation criterion given to the best subset found using forward selection.

```
if (!selector.fail()) {
    cout << "Selected variables: " << selector.selectedParamIndices();
    cout << "Selected parameter values: "
         << selector.selectedParamValues();
    cout << "Subset evaluation value: "
         << selector.evalFunctionForSelected();
}
```

Now we switch to using stepwise selection and see if a better subset is found.

```
selector.setSearchMethod(rwStepwiseSelection);
cout << "New subset evaluation value: "
<< selector.evalFunctionForSelected();
```

By comparing the results of the linear regression using forward selection to the results using stepwise selection, you will find the subset of variables that should be used for prediction.

Summary

Analytics.h++ provides a large, powerful collection of classes that simplify the development and maintenance of business intelligence applications. You will easily manipulate and preprocess business data, form predictive models, and perform in-depth analysis of the models within a high-level, easy-to-learn class structure that is optimized for high-speed processing on the most popular hardware platforms. You can still use your legacy code from C and FORTRAN if desired, easily migrating to C++ without sacrificing your previous investment.

This report has focused on some valuable features of Analytics.h++, but the discussion of this deeply functional product is by no means complete. There are many features, classes, and issues that we couldn't include in this brief format. Should you have questions about how Analytics.h++ would function in your development environment, Rogue Wave's technical engineers and account representatives are available to assist you. Please refer to the back of this report for information on how to contact the office nearest you.

This version of Analytics.h++ is the first in a comprehensive business analysis library. We plan to add more algorithms in the future, including those for optimization, neural networks, statistical population comparison, outlier detection, and more. Our development teams are interested in creating libraries that serve your needs, so please let your Rogue Wave account representative know if you have any specific algorithms you wish to have included in future versions.

© Copyright Rogue Wave Software, Inc. 1998. All Rights Reserved.
Rogue Wave and .h++ are registered trademarks of Rogue Wave Software, Inc.
All other trademarks are the property of their respective holders.

Corporate Headquarters
Toll-free: (800) 487-3217
Email: sales@roguewave.com

Rogue Wave Software B.V.
Telephone: +31-20-301 26 26
www.roguewave.nl

Rogue Wave Software GmbH
Telephone: +49-6103-59 34-0
www.roguewave.de

Rogue Wave Software S.A.R.L.
Telephone: +33-1-4196 2626
www.roguewave.fr

Rogue Wave Software-UK.Ltd.
Telephone: +44 (0) 118 9358600
www.roguewave.co.uk

Rogue Wave Software S.R.L.
Telephone: +39-02-4125 081
www.roguewave.it

