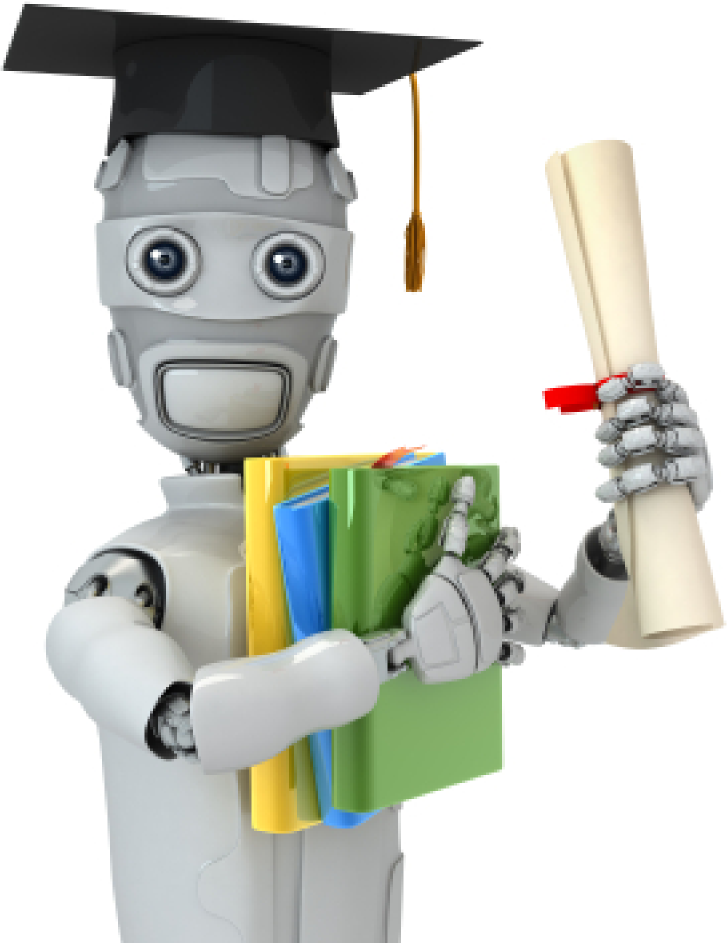
HPCC Systems™

Machine Learning Library Reference

Boca Raton Documentation Team



Machine Learning Library Reference

Boca Raton Documentation Team

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com> Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Version Number in the text of the message.

July 2013 Version 1.2.0C

Introduction and Installation

LexisNexis Risk Solutions is an industry leader in data content, data aggregation, and information services which has independently developed and implemented a solution for data-intensive computing called HPCC (High-Performance Computing Cluster).

The HPCC System is designed to run on clusters, leveraging the resources across all nodes. However, it can also be installed on a single machine and/or VM for learning or test purposes. It will also run on the AWS Cloud.

The HPCC platform also includes ECL (Enterprise Control Language) which is a powerful high-level, heavily-optimized, data-centric declarative language used for parallel data processing. The flexibility of the ECL language is such that any ECL code will run unmodified regardless of the size of the cluster being used.

Instructions for installing the HPCC are available on the HPCC Systems website, http://hpccsystems.com/community/docs/installation-and-administration. More information about running HPCC on the AWS Cloud can be found in this document, *Running the HPCC System's Thor Platform within Amazon Web Services*, http://hpccsystems.com/community/docs/aws-install-thor.

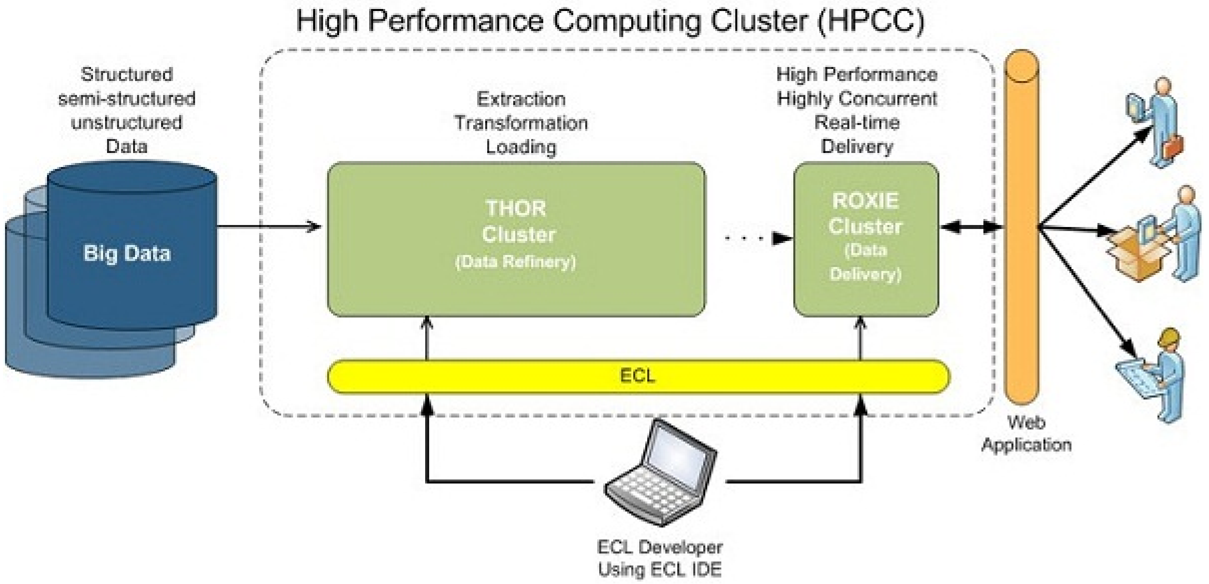
The HPCC Platform

HPCC is fast, flexible and highly scalable. It can be used for any data-centric task and can meet the needs of any database regardless of size. There are two types of cluster:

- The Thor cluster is used to process all data in all files.
- The Roxie cluster is used to search for a particular record or set of records.

As shown by the following diagram, the HPCC architecture also incorporates:

- Common middle-ware components.
- An external communications layer.
- Client interfaces which provide both end-user services and system management tools.
- Auxiliary components to support monitoring and to facilitate loading and storing of file system data from external sources



For more information about the HPCC architecture, clusters and components, see the HPCC website, http://hpccsystems.com/Why-HPCC/How-it-works.

The ECL Programming Language

The ECL programming language is a key factor in the flexibility and capabilities of the HPCC processing environment. It is designed to be a transparent and implicitly parallel programming language for data-intensive applications. It is a high-level, highly-optimized, data-centric declarative language that allows programmers to define what the data processing result should be and the dataflows and transformations that are necessary to achieve the result.

Execution is not determined by the order of the language statements, but from the sequence of dataflows and transformations represented by the language statements. It combines data representation with algorithm implementation, and is the fusion of both a query language and a parallel data processing language.

ECL uses an intuitive syntax which has taken cues from other familiar languages, supports modular code organization with a high degree of reusability and extensibility, and supports high-productivity for programmers in terms of the amount of code required for typical applications compared to traditional languages like Java and C++. It is compiled into optimized C++ code for execution on the HPCC system platform, and can be used for complex data processing and analysis jobs on a Thor cluster or for comprehensive query and report processing on a Roxie cluster.

ECL allows inline C++ functions to be incorporated into ECL programs, and external programs in other languages can be incorporated and parallelized through a PIPE facility.

External services written in C++ and other languages which generate DLLs can also be incorporated in the ECL system library, and ECL programs can access external Web services through a standard SOAPCALL interface.

The ECL language includes extensive capabilities for data definition, filtering, data management, and data transformation, and provides an extensive set of built-in functions to operate on records in datasets which can include user-defined transformation functions. The Thor system allows data transformation operations to be performed either locally on each node independently in the cluster, or globally across all the nodes in a cluster, which can be user-specified in the ECL language.

An additional important capability provided in the ECL programming language is support for natural language processing (NLP) with PATTERN statements and the built-in PARSE operation. Using this capability of the ECL language it is possible to implement parallel processing from information extraction applications across document files including XML-based documents or Web pages.

Some benefits of using ECL are:

- It incorporates transparent and implicit data parallelism regardless of the size of the computing cluster and reduces the complexity of parallel programming increasing development productivity.
- It enables the implementation of data-intensive applications with huge volumes of data previously thought to be intractable or infeasible. ECL was specifically designed for manipulation of data and query processing. Orders of magnitude performance increases over other approaches are possible.
- The ECL compiler generates highly optimized C++ for execution.
- It is a powerful, high-level, parallel programming language ideal for implementation of ETL, information retrieval, information extraction, record linking and entity resolution, and many other data-intensive applications.
- It is a mature and proven language but is still evolving as new advancements in parallel processing and data intensive computing occur.

The HPCC platform also provides a comprehensive IDE (ECL IDE) which provide a highly interactive environment for rapid development and implementation of ECL applications.

HPCC and the ECL IDE downloads are available from the HPCC systems website, http://hpccsystems.com/ which also provides access to documentation and tutorials.

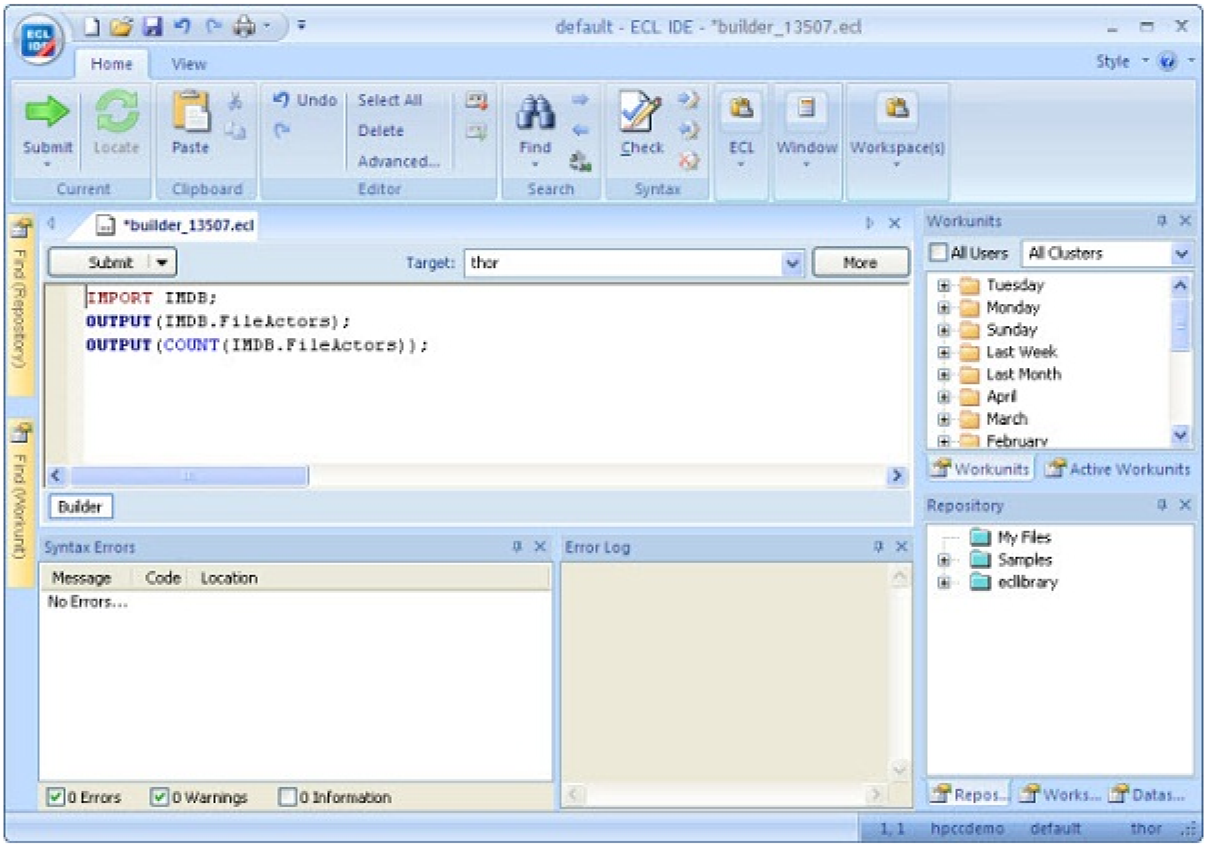
ECL IDE

ECL IDE is an ECL programmer's tool. Its main use is to create queries and ECL files and is designed to make ECL coding as easy as possible. It has all the ECL built-in functions available to you for simple point-and-click use in your query construction. For example, the Standard String Library (Std.Str) contains common functions to operate on STRING fields such as the ToUpperCase function which converts characters in a string to uppercase.

You can mix-and-match your data with any of the ECL built-in functions and/or ECL files you have defined to create Queries. Because ECL files build upon each other, the resulting queries can be as complex as needed to obtain the result.

Once the Query is built, submit it to an HPCC cluster, which will process the query and return the results.

Configuration files (.CFG) are used to store the information for any HPCC you want to connect to, for example, it stores the location of the HPCC and the location of any folders containing ECL files that you may want to use while developing queries. These folders and files are shown in the **Repository** window.



Target: thor

```
IMPORT IMDB;
OUTPUT(IMDB.FileActors);
OUTPUT(COUNT(IMDB.FileActors));
```

Message	Code	Location
No Errors...		

☑ 0 Errors ☑ 0 Warnings □ 0 Information

1, 1 hpccdemo default thor

For more information on using ECL IDE see the Client Tools manual which may be downloaded from the HPCC website, http://hpccsystems.com/community/docs/client-tools.

Installing and using the ML Libraries

The ML Libraries can only be used in conjunction with an HPCC System, ECL IDE and the Client tools.

Requirements

If you don't already use the HPCC platform and/or ECL IDE and the Client Tools, you must download and install them before downloading the ML libraries:

- Download and install the relevant HPCC platform for your needs. (http://hpccsystems.com/download/free-community-edition)
- Download and install the ECL IDE and Client Tools. (http://hpccsystems.com/download/free-community-edition/ecl-ide-and-client-tools)

The ML Libraries can also be used on an HPCC Systems One-Click^{TM} Thor, which is available to anyone with an Amazon AWS account. To walk-through an example of how to use the One-Click^{TM} Thor with the Machine Learning Libraries, see the *Associations (ML.Assocaite)* section in *The ML Module* chapter later in this manual.

To setup a One-Click^{TM} Thor cluster:

- Setup an Amazon AWS account (http://aws.amazon.com/account/)
- Login and Launch your Thor cluster. The **Login** button on the HPCC Systems website (https://aws.hpccsystems.com/aws/getting_started/), provides an automated setup process which is quick and easy to use.
- Download the ECL IDE and Client Tools onto your computer ((http://hpccsystems.com/download/free-community-edition/ecl-ide-and-client-tools))

If you are new to the ECL Language, take a look at the a programmers guide and language reference guides, http://hpccsystems.com/community/docs/learning-ecl.

The HPCC Systems website also provides tutorials designed to get you started using data on the HPCC System, http://hpccsystems.com/community/docs/tutorials.

Installing the ML Libraries

To install the ML Libraries:

1. Go to the Machine Learning page of the HPCC Systems website, http://hpccsystems.com/ml and click on **Download and Get Started**.
2. Click on **Step 1: Download the ML Library** and save the file to your computer.
3. Extract the downloaded files to your ECL IDE source folder. This folder is typically located here: "C:\Users\Public\Documents\HPCC Systems\ECL\My Files".

Note: To find out the location of your **Working Folder**, simply go to your ECL IDE **Preferences** window either from the login dialog or from the **Orb** menu. Click on the **Compiler** tab and use the first **Working Folder** location listed.

The ML Libraries are now ready to be used. To locate them, display the **Repository Window** in ECL IDE and expand the **My Files** folder to see the **ML** folder.

Using the ML Libraries

A walk-through is provided for all Machine Learning Libraries supported, which are designed to get you started using ML with the HPCC System. Each module is also covered in this manual in a separate section which contains more detailed information about the functionality of the routines included.

To use the ML Libraries, you also need to upload some data onto the **Dropzone** of your cluster. If you already have a file on your computer, you can upload it onto the **Dropzone** using **ECL Watch**. Simply, use the **DFU Files/ Upload/download** menu item, locate the file(s), select and upload.

Now that the ML Libraries are installed and you have uploaded your data, you can use ECL IDE to write queries to analyze your data:

1. Login to ECL IDE, accessing the HPCC System you have installed.

2. Using the **Repository** toolbox, expand **My Files**.

3. Expand the **ML** folder to locate the Machine Learning files you want to use.

4. Open a new builder window and start writing your query. To reference the ML libraries in your ECL source code, use an import statement. For example:

```
IMPORT * FROM ML;
IMPORT * FROM ML.Cluster;
IMPORT * FROM ML.Types;

//Define my record layout
MyRecordLayout := RECORD
UNSIGNED RecordId;
REAL XCoordinate;
REAL YCoordinate;
END;

//My dataset
X2 := DATASET([
{1, 1, 5},
{2, 5, 7},
{3, 8, 1},
{4, 0, 0},
{5, 9, 3},
{6, 1, 4},
{7, 9, 4}], MyRecordLayout);

//Three candidate centroids
CentroidCandidates := DATASET([
{1, 1, 5},
{2, 5, 7},
{3, 9, 4}], MyRecordLayout);

//Convert them to our internal field format
ml.ToField(X2, fX2);
ml.ToField(CentroidCandidates, fCentroidCandidates);

//Run K-Means for, at most, 10 iterations and stop if delta < 0.3 between iterations
fX3 := Kmeans(fX2, fCentroidCandidates, 10, 0.3);

//Convert the final centroids to the original layout
ml.FromField(fX3.result(), MyRecordLayout, X3);

//Display the results
OUTPUT(X3);
```

Contributing to the sources

Both HPCC and ECL-ML are open source projects and contributions to the sources are welcome. If you are interested in contributing to these projects, simply download the GitHub client and go to the relevant GitHub pages.

- To contribute to the HPCC open source project, go to https://github.com/hpcc-systems/HPCC-Platform.
- To contribute to the ECL-ML open source project, go to https://github.com/hpcc-systems/ecl-ml.

You are required to sign a contribution agreement to become a contributor.

Machine Learning Algorithms

The HPCC Systems Machine Learning libraries contain an extensible collection of machine learning routines which are easy and efficient to use and are designed to execute in parallel across a cluster. T

he list of modules supported will continue to grow over time. The following modules are currently supported:

- Associations (ML.Associate)
- Classify (ML.Classify)
- Cluster (ML.Cluster)
- Correlations (ML.Correlate)
- Discretize (ML.Discretize)
- Distribution (ML.Distribution)
- Field Aggregates (ML.FieldAggregates)
- Regression (ML.Regression)
- Visualization (ML.VL)

The Machine Learning modules are supported by the following which are also used to implement ML:

- The Matrix Library (Mat)
- Utility (ML.Utility)
- Docs (ML.Doc)

The ML Modules are used in conjunction with the HPCC system. More information about the HPCC System is available on the following website, http://hpccsystems.com/.

The ML Data Models

The ML routines are all centered around a small number of core processing models. As a user of ML (rather than an implementer) the exact details of these models can generally be ignored. However, it is useful to have some idea of what is going on and what routines are available to help you with the various models. The formats that are shared between various modules within ML are all contained within the Type definition.

Numeric field

The principle type that undergirds most of the ML processing is the Numeric Field. This is a general representation of an arbitrary ECL record of numeric entries. The record has 3 fields:

Field	Description
Id	The 'record' id. This is an identifier for the record being modeled. It will be shared between all of the fields of the record.
Field Number	And ECL record with 10 fields produces 10 ‘numericfield’ records, one with each of the field numbers from 1 to 10[1].
Value	The value of the field.

This is perhaps visualized by comparison to a traditional ECL record. Here is a simple example showing some height, weight and age facts for certain individuals:

```
IMPORT ml;

value_record := RECORD
                UNSIGNED rid;
                REAL height;
                REAL weight;
                REAL age;
                INTEGER1 species; // 1 = human, 2 = tortoise
                INTEGER1 gender; // 0 = unknown, 1 = male, 2 = female
END;


d := dataset([{1,5*12+7,156*16,43,1,1},
              {2,5*12+7,128*16,31,1,2},
              {3,5*12+9,135*16,15,1,1},
              {4,5*12+7,145*16,14,1,1},
              {5,5*12-2,80*16,9,1,1},
              {6,4*12+8,72*16,8,1,1},
              {7,8,32,2.5,2,2},
              {8,6.5,28,2,2,2},
              {9,6.5,28,2,2,2},
              {10,6.5,21,2,2,1},
              {11,4,15,1,2,0},
              {12,3,10.5,1,2,0},
              {13,2.5,3,0.8,2,0},
              {14,1,1,0.4,2,0}
              ]
              ,value_record);


d;
```

It has 14 rows of data. Each row has 5 interesting data fields and a *record id* that is prepended to uniquely identify the record. Therefore a *5 field ECL record* actually has 6 fields.

ML provides the ToField operation that converts a record in this general format to the NumericField format. Thus:

```
ml.ToField(d,o);
d;
o
```

Shows not only the original data, but also the data in the standard ML NumericField format. The latter has 70 rows (5x14). Incidentally: ToField is an example of a macro that uses a 'out' parameter (o) rather than returning a value. If a file has N rows and M columns then the order of the ToField operation will be O(mn). It is also possible to turn the NumericField format back into a *regular* ECL style record using the FromField operation:

```
ml.ToField(d,o);
d;
o;
ml.FromField(o,value_record,d1);
d1;
```

Will leave d1 = d.

Advanced - Converting more complex records

By default, the ToField operation assumes the first field is the “id” field, and all subsequent numeric fields are to be assigned a field number in the resulting table. However, additional parameters may be specified to ToField that facilitates the ability to specify the name of the id column in the original table as well as the columns to be used as data fields. For example:

```
IMPORT ML;
value_record := RECORD
STRING first_name;
STRING last_name;
UNSIGNED name_id;
REAL height;
REAL weight;
REAL age;
STRING eye_color;
INTEGER1 species; // 1 = human, 2 = tortoise
INTEGER1 gender; // 0 = unknown, 1 = male, 2 = female
END;

dOrig := dataset([
{'Charles','Babbage',1,5*12+7,156*16,43,'Blue',1,1},
{'Tim','Berners-Lee',2,5*12+7,128*16,31, 'Brown',1,1},
{'George','Boole',3,5*12+9,135*16,15, 'Hazel',1,1},
{'Herman','Hollerith',4,5*12+7,145*16,14,'Green',1,1},
{'John','Von Neumann',5,5*12-2,80*16,9,'Blue',1,1},
{'Dennis','Ritchie',6,4*12+8,72*16,8, 'Brown',1,1},
{'Alan','Turing',7,8,32,2.5, 'Brown',2,1}
],value_record);

ML.ToField(dOrig,dResult,name_id,'height,weight,age,gender');
dOrig;
dResult;
```

In the above example, the name_id column is taken as the id. Height, weight, age and gender will be parsed into numbered fields.

Note: The id name is not in quotes, but the comma-delimited list of fields is.

Along with creating the new table in NumericField format, the ToField macro also creates three other objects to help with field translation, two functions and a dataset.

The two functions are outtable_ToName() and outtable_ToNumber(), where outtable is the name of the output table specified in the macro call. Passing in a number in the first one will produce the field name mapped to that number, and passing a string into the second one will produce the number assigned to that field name.

For the previous example, we can therefore do the following:

```
dResult_ToName(2);      // Returns ‘weight’
dResult_ToNumber(‘age’) // Returns 3 (note that the field name is always lowercase)
```

The other dataset that is created is a 2-column mapping table named outtable_Map which contains every field from the original table in the first column, and what it is mapped to in the second column.

This would either be the column number, the string “ID” if it is the ID field, or the string “NA” indicating that the field was not mapped to a NumericField number. In the above example, the table is named:

```
dResult_Map;
```

The mapping table may be used when reconstituting the data back to the original format. For example:

```
ML.FromField(dResult,value_record,dReconstituted,dResult_Map);
dReconstituted;
```

The output from this FromField call will have the same structure as the initial table, and values that existed in the NumericField version of the table will be allocated to the fields specified in the mapping table.

Note: Any data that did not translate into the NumericField table will be left blank or zero in the reconstituted table.

Discrete field

Some of the ML routines do not require the field values to be real, rather they require discrete (integral) values. The structure of the records are essentially identical to NumericField but, the value is of type t_Discrete (typically INTEGER) rather than t_FieldReal (typically REAL8).

There are no explicit routines to get to a discrete-field structure from an ECL record, rather it is presumed that NumericField will be used as an intermediary.

There is an entire module (Discretize) devoted to moving a NumericField structured file into a DiscreteField structured file. The options and reasons for the options are described in the Discretize module section.

For this introduction it is adequate to show that all of the numeric fields could be made integral simply by using:

```
ml.ToField(d,o);
o;
o1 := ML.Discretize.ByRounding(o);
o1
```

ItemElement

A rather more specialist format is the ItemElement format. This does not model an ECL record directly, rather it models an abstraction that can be derived from an ECL record.

The item element has a record id and a value (which is of type t_Item). The t_Item is an integral value – but unlike t_Discrete the values are not considered to be ordinal. Put another way, in t_Discrete 4 > 3 and 2 < 3. In t_Item the 2, 3, 4 are just arbitrary labels that ‘happen’ to be integers for efficiency.

Note: ItemElement does not have a field number.

There is no significance placed upon the field from which the value was derived. This models the abstract notion of a collection of ‘bags’ of items. An example of the use of this type of structure will be given in the Using ML with documents section.

Coding with the ML data models

The ML data models are extremely flexible to work with; but using them is a little different from traditional ECL programming. This section aims to detail some of the possibilities.

Column splitting

Some of the ML routines expect to be handed two datasets which may be, for example, a dataset of independent variables and another of dependent variables. The data as it originally exists will usually have the independent and dependent data within the same row. For example, when using a classifier to produce a model to predict the species or gender of an entity from the other details, the height, weight and age fields would need to be in a different ‘file’ to the species and gender. However, they have to have the same record ID to show the correlation between the two. In the ML data model this is as simple as applying two filters:

```
ml.ToField(d,o);
o1  := ML.Discretize.ByBucketing(o,5);
Independents := o1(Number <= 3);
Dependents := o1(Number >= 4);
Bayes := ML.Classify.BuildNaiveBayes(Independents,Dependents);
Bayes
```

Genuine nulls

Implementing a genuine null can be done by simply removing certain fields with certain values from the datastream. For example, if 0 was considered an invalid weight then one could do:

```
Better := o(Number<>2 OR Value<>0);
```

Sampling

By far the easiest way to split a single data file into samples is to use the SAMPLE and ENTH verbs upon the datafile PRIOR to the conversion to ML format.

Inserting a column with a computed value

Inserting a column with a new value computed from another field value is a fairly advanced technique. The following inserts the square of the weight as a new column:

```
ml.ToField(d,o);

BelowW := o(Number <= 2);
// Those columns whose numbers are not changed
// Shuffle the other columns up - this is not needed if appending a column
AboveW := PROJECT(o(Number>2),TRANSFORM(ML.Types.NumericField,SELF.Number :=
LEFT.Number+1, SELF := LEFT));
NewCol := PROJECT(o(Number=2),TRANSFORM(ML.Types.NumericField,
                                                        SELF.Number := 3,
                                                        SELF.Value := LEFT.Value*LEFT.Value,
                                                        SELF := LEFT) );

NewO := BelowW+AboveW+NewCol;

NewO;
```

Generating test data

ML is interesting when it is being executed against data with meaning and significance. However, sometimes it can be useful to get hold of a lot of data quickly for testing purposes. This data may be ‘random’ (by some definition) or it may follow a number of carefully planned statistical distributions. The ML libraries have support for high performance ‘random value’ generation using the GenData command inside the distribution module.

GenData generates one column at a time although it generates that column for all the records in the file. It works in parallel so is very efficient.

The easiest type of column to generate is one in which the values are evenly and randomly distributed over a range. The following generates 1M records each with a random number from 0-100 in the first column:

```
IMPORT ML;

TestSize := 1000000;
a1 := ML.Distribution.Uniform(0,100,10000);
ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
```

To generate 1M records with three columns; one Uniformly distributed, one Normally distributed (mean 0, Standard Deviation 10) and one with a Poisson distribution (Mean of 4):

```
IMPORT ML;

TestSize := 1000000;

a1 := ML.Distribution.Uniform(0,100,10000);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
// Field 2 Normally Distributed
a2 := ML.Distribution.Normal2(0,10,10000);
b2 := ML.Distribution.GenData(TestSize,a2,2);
// Field 3 - Poisson Distribution
a3 := ML.Distribution.Poisson(4,100);
b3 := ML.Distribution.GenData(TestSize,a3,3);

D := b1+b2+b3; // This is the test data

ML.FieldAggregates(D).Simple;  // Perform some statistics on the test data to ensure
                                  it worked
```

This generates the data in the correct format and even produces some statistics to ensure it works!

The ML libraries have over half a dozen different distributions that the generated data columns can be given. These are described at length in the Distribution module section.

ML module walk-throughs

To help you get started, a walk-through is provided for each ML module. The walk-throughs explain how the modules work and demonstrate how they can be used to generate the results you require.

Association walk-through

Association mining is one of the most wide-spread, if not widely known forms of machine learning. If you have ever entered a few items into an online ‘shopping-basket’ and then been prompted to buy more things, which were exactly what you wanted, then the chances are there was an association miner working in the background.

At their simplest, association mining algorithms are handed a large number of ‘collections of items’ and then they find which items co-occur in most of the collections. The ECL-ML association algorithms are used by instantiating an Association module passing in a dataset of Items and a number, which is the minimum number of co-occurrences considered to be significant (the lower this number, the slower the algorithm).

The following code creates such a module using data which is randomly generated but tweaked to have some relationships within it.

```
IMPORT ML;
TestSize := 100000;
CoOccurs := TestSize/1000;

a1 := ML.Distribution.Poisson(5,100);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
a2 := ML.Distribution.Poisson(3,100);
b2 := ML.Distribution.GenData(TestSize,a2,2);
a3 := ML.Distribution.Poisson(3,100);
b3 := ML.Distribution.GenData(TestSize,a3,3);

D := b1+b2+b3; // This is the test data
// Now construct a fourth column which is a function of column 1
B4 := PROJECT(b1,TRANSFORM(ML.Types.NumericField, SELF.Number:=4, SELF.Value:=LEFT.Value * 2,
      SELF.Id := LEFT.id));

AD0 := PROJECT(ML.Discretize.ByRounding(B1+B2+B3+B4),ML.Types.ItemElement);
// Remove duplicates from bags (fortunately the generation allows this to be local)
AD := DEDUP( SORT( AD0, ID, Value, LOCAL ), ID, Value, LOCAL );

ASSO := ML.Associate(AD,CoOccurs);
```

The simplest question which can now be asked is: “Which pairs of items are most likely to appear together?”. The following provides the answer:

```
TOPN(Asso.Apriori2,50,-Support)
```

The answer to: ‘Which triplets can be found together?’ is answered by:

```
TOPN(Asso.Apriori3,50,-Support);
```

The same answer can also be found by asking the question:

```
TOPN(Asso.EclatN(3,3),50,-Support);
```

The second form uses a different algorithm for the computation (Eclat), which also has a more flexible interface. The first parameter gives the maximum group size that is interesting. The second parameter gives the minimum group size that is interesting.

Thus to find the largest groups of size 4 or 3 use:

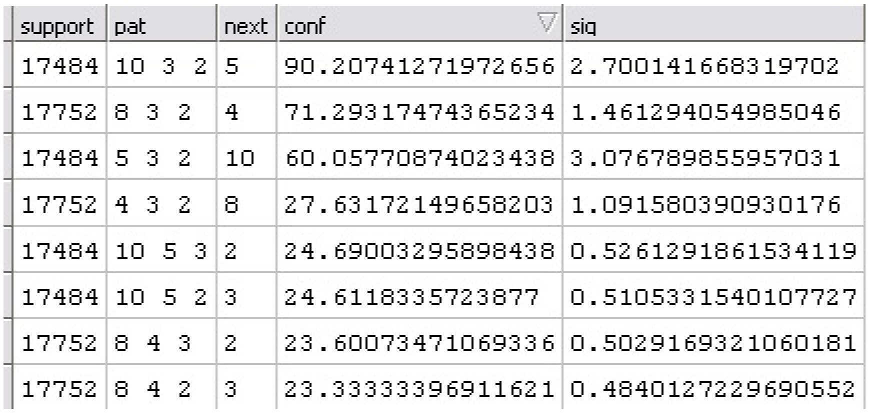
```
TOPN(Asso.EclatN(4,3),50,-Support);
```

It may be noted that there is also an AprioriN function. However it relies upon a feature that is not currently functional in version 3.4.2 of the HPCC platform.

In addition to being able to spot the common patterns, the association module is able to turn a set of patterns into a set of rules or to use a different term of art, it is capable of building a predictor. Essentially a predictor answers the question: “Given I have this in my basket, what will come next.?”. A predictor is built by passing the output of EclatN or AprriorN into the Rules function:

```
R := TOPN(Asso.EclatN(4,2),50,-Support);

Asso.Rules(R)
```

This produces the following numbers:



support	pat	next	conf	sig
17484	10 3 2	5	90.20741271972656	2.700141668319702
17752	8 3 2	4	71.29317474365234	1.461294054985046
17484	5 3 2	10	60.05770874023438	3.076789855957031
17752	4 3 2	8	27.63172149658203	1.091580390930176
17484	10 5 3	2	24.69003295898438	0.5261291861534119
17484	10 5 2	3	24.6118335723877	0.5105331540107727
17752	8 4 3	2	23.60073471069336	0.5029169321060181
17752	8 4 2	3	23.33333396911621	0.4840127229690552

Note: Your numbers will be different because the data is randomly generated when run.

The support tells you how many patterns were used to make the prediction.

Conf tells the percentage of times that the prediction would be correct (in the training data, but real life might be different!).

Sig is used to indicate whether the next item is likely to be causal (high sig) or co-incidental (low-sig). To understand the difference, imagine that you go into Best Buy to buy an expensive telephone. Your shopping basket of 1 item will probably allow the system to predict two different likely next items, such as a case for the phone and a candy bar. They might both have high confidence, but the case will have high significance (you will usually by the case if you by the phone), the candy will not (it is only likely because ‘everyone buys candy’).

Classification walk-through

Modules: Classify

ML.Classify tackles the problem: “given I know these facts about an object; can I predict some other value or attribute of that object.” This is really where data processing gives way to machine learning: based upon some form of training set can I derive a rule or model to predict something something about other data records.

Classification is sufficiently central to machine learning that we provide four different methods of doing it. You will need to examine the literature or experiment to decide exactly which method of classification will work best in any given context. In order to simplify coding and to allow experimentation all of our classifiers can be used through the unified classifier interface.

Using a classifier in ML can be viewed as three logical steps:

1. Learning the model from a training set of data that has been classified externally.

2. Testing. Getting measures of how well the classifier fits.

3. Classifying. Apply the classifier to new data in order to give it a classification.

In the examples that follow we are simply trying to show how a given method can be used in a given context; we are not necessarily claiming it is the best or only way to solve the given example.

A classifier will not predict anything if handed totally random data. It is precisely looking for a relationship between the data that is non-random. So this example will generate test data by:

1. Generating three random columns.

2. Producing a fourth column that is the sum of the three columns.

3. Giving the fourth column a category from 0 (small) to 2 (big).

The object is to see if the system can learn to predict from the individual fields which category the record will be assigned. The data generation is a little more complex than normal so it is presented here (this code is also available in Tests/Explanatory/Classify in our source distribution).

```
IMPORT ML;
// First auto-generate some test data for us to classify

TestSize := 100000;
a1 := ML.Distribution.Poisson(5,100);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
a2 := ML.Distribution.Poisson(3,100);
b2 := ML.Distribution.GenData(TestSize,a2,2);
a3 := ML.Distribution.Poisson(3,100);
b3 := ML.Distribution.GenData(TestSize,a3,3);

D := b1+b2+b3; // This is the test data

// Now construct a fourth column which is the sum of them all
B4 := PROJECT(TABLE(D,{Id,Val := SUM(GROUP,Value)},Id),TRANSFORM(ML.Types.NumericField,
                                           SELF.Number:=4,
                                           SELF.Value:=MAP(LEFT.Val < 6 => 0,  // Small
                                                           LEFT.Val < 10 => 1, // Normal
                                                                          2 ); // Big
                                           SELF := LEFT));
D1 := D+B4;
```

The data generated for D1 is in numeric field format which is to say that the variables are continuous (real numbers). Classifiers require that the 'target' results (the numbers to be produced) are positive integers. The unified classier interface allows for the inputs to a classifier to be either continuous or discrete (using the 'C' and 'D' versions of the functions). However, most of the implemented classifiers prefer discrete input and you get better control over the discretization process if you do it yourself. The Discretize module can do this (see the section ML Data Models and the Discretize module which explain this in more detail) but to keep things simple we will just round all the fields to integers:

```
// We are going to use the 'discrete' classifier interface, so discretize our data first
D2 := ML.Discretize.ByRounding(D1);
```

In the rest of this section if a 'D2' appears from 'nowhere', it is referencing this dataset.

Every classifier has a module within the classify module. It is usually worth grabbing hold of that first to make the rest of the typing simpler. In this case we will get the NaiveBayes module:

```
BayesModule := ML.Classify.NaiveBayes;
```

While I labeled it 'BayesModule', it important to understand that that one line is the only difference between whether you are using NaiveBayes, Perceptrons, LogisticRegression or one of the other classifier mechanisms.

For illustration purposes we will skip straight to testing:

```
TestModule := BayesModule.TestD(D2(Number<=3),D2(Number=4));
TestModule.Raw;
TestModule.CrossAssignments;
TestModule.PrecisionByClass;
TestModule.Headline;
```

The TestModule := does all the work.

Firstly note that D2 has been split into two pieces. The first parameter is all of the independent variables (sometimes called features), the second parameter is the dependent variables (or classes). The fact that TestD was called (rather than TestC) is to indicate that the independent variables are discrete.

The module now has four different outputs to show you how well the classification worked:

Result	Description
Headline	Gives you the main precision number. On this test data, the result shows how often the classifier was correct.
PrecisionByClass	Similar to Headline except that it gives the precision broken down by the class that it SHOULD have been classified to. It is possible that a classifier might work well in general but may be particularly poor at identifying one of the groups.
CrossAssignments	It is one thing to say a classification is 'wrong'. This table shows, "if a particular class is mis-classified, what is it most likely to be mis-classified as?".
Raw	Gives a very detailed breakdown of every record in the test corpus, for example, what the classification should have been and what it was.

Assuming you like the results you will normally learn the model and then use it for classification. In the 'real world' you would probably do the learning and the classifying at very different times and on very different data. For illustration purposes and simplicity this code learns the model and uses it immediately on the same data:

```
Model := BayesModule.LearnD(D2(Number<=3),D2(Number=4));
Results := BayesModule.ClassifyD(D2(Number<=3),Model);
Results;
```

Logistic Regression

Regression analysis includes techniques for modeling the relationship between a dependent variable Y and one or more independent variables Xi.

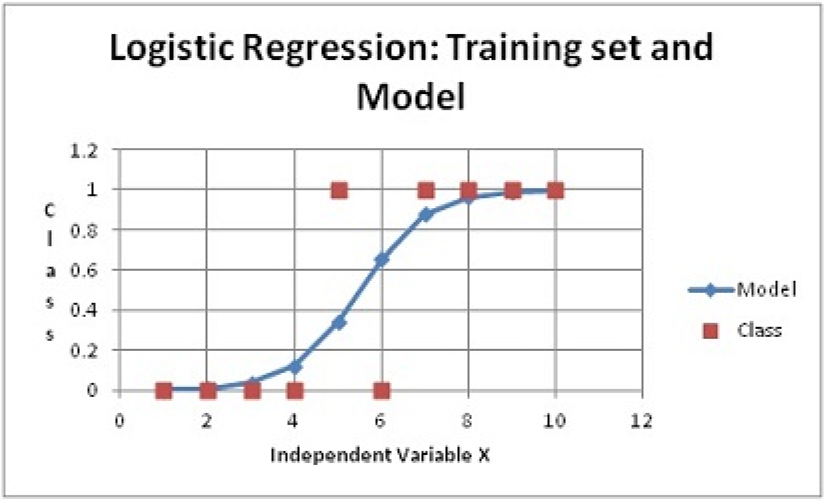
The most common form of regression model is the Ordinary Linear Regression (OLR) which fits a line through a set of data points.

While the linear regression model is simple and very applicable to many cases, it is not adequate for some purposes. For example, if dependent variable Y is binary, i.e. if Y takes either 0 or 1, then a linear model, which has no bounds on what values the dependent variable Y can take, cannot represent the relationship between X and Y correctly. In that case, the relationship can be modeled using a logistic function, also known as sigmoid function, which is an S-shaped curve with values from (0,1). Since dependent variable Y can take only two values, 0 or 1, the Logistic Regression model predicts two outcomes, 0 or 1, and it can be used as a tool for classification.

For example, given the following data set:

X	**Y**
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	1
9	1
10	1

The Logistic Regression produces the model below:

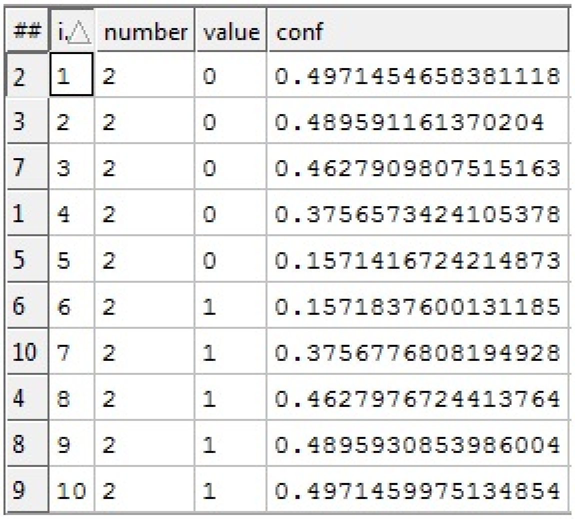


This model is then used as a classifier which assigns class 0 to every point xi where yi<0.5, and class 1 for every xi where yi>=0.5.

In the following example we have a dataset equivalent to the dataset used to create the Logistic Regression model depicted above.

```
IMPORT ML;

value_record := RECORD
                UNSIGNED rid;
                REAL length;
                INTEGER1 class;
END;

d := DATASET([{1,1,0}, {2,2,0}, {3,3,0}, {4,4,0}, {5,5,1},
              {6,6,0}, {7,7,1}, {8,8,1}, {9,9,1}, {10,10,1}]
             ,value_record);

ML.ToField(d,o);
Y := O(Number=2);  // pull out class
X := O(Number=1); // pull out lenghts

dY := ML.Discretize.ByRounding(Y);
LogisticModule := ML.Classify.Logistic(,,10);
Model := LogisticModule.LearnC(X,dY);
LogisticModule.ClassifyC(X,Model);
```

The classifier produces the following result:



##	i△	number	value	conf
2	1	2	0	0.4971454658381118
3	2	2	0	0.489591161370204
7	3	2	0	0.4627909807515163
1	4	2	0	0.3756573424105378
5	5	2	0	0.1571416724214873
6	6	2	1	0.1571837600131185
10	7	2	1	0.3756776808194928
4	8	2	1	0.4627976724413764
8	9	2	1	0.4895930853986004
9	10	2	1	0.4971459975134854

As expected, the independent variable values 5 and 6 have been mis-classified compared to the training set, but the confidence in those classification results is low. As depicted in the logistic regression figure above, mis-classification happens because the model function value for x=5 is less than 0.5 and it is greater than 0.5 for x=6.

Cluster Walk-through

Modules: Cluster, Doc

The cluster module contains routines that can be used to find groups of records that appear to be ‘fairly similar’.

The module has been shown to work on records with as few as two fields and as many as sixty thousand. The latter was used for clustering documents of words (see Using ML with documents). The clustering module has more than half a dozen different ways of measuring the distance (defining ‘similar’) between two records but it is also possible to write your own.

Below are walk-throughs for the methods covered in the ML.Cluster module. Each begins with the following set of entities in 2-dimensional space, where the values on each axis are restricted to between 0.0 and 10.0:

```
IMPORT ML;
lMatrix:={UNSIGNED id;REAL x;REAL y;};

dEntityMatrix:=DATASET([
  {1,2.4639,7.8579},
  {2,0.5573,9.4681},
  {3,4.6054,8.4723},
  {4,1.24,7.3835},
  {5,7.8253,4.8205},
  {6,3.0965,3.4085},
  {7,8.8631,1.4446},
  {8,5.8085,9.1887},
  {9,1.3813,0.515},
  {10,2.7123,9.2429},
  {11,6.786,4.9368},
  {12,9.0227,5.8075},
  {13,8.55,0.074},
  {14,1.7074,3.9685},
  {15,5.7943,3.4692},
  {16,8.3931,8.5849},
  {17,4.7333,5.3947},
  {18,1.069,3.2497},
  {19,9.3669,7.7855},
  {20,2.3341,8.5196}
],lMatrix);
ML.ToField(dEntityMatrix,dEntities);
```

Note: The use of the ToField macro which converts the original rectangular matrix, dEntityMatrix, into a table in the standard NumericField format that is used by the ML library named “dEntities”.

KMeans

With k-means clustering the user creates a second set of entities called centroids, with coordinates in the same space as the entities being clustered. The user defines the number of centroids (k) to create, which will remain constant during the process and therefore represents the number of clusters that will be determined. For our example, we will define four centroids:

```
dCentroidMatrix:=DATASET([
  {1,1,1},
  {2,2,2},
  {3,3,3},
  {4,4,4}
],lMatrix);

ML.ToField(dCentroidMatrix,dCentroids);
```

As with the entity matrix, we have used ToField to convert the centroid matrix into the table "dCentroids".

Note: Although these points are arbitrary, they are clearly not random.

These points form an asymmetrical pattern in one corner of the space. This is to highlight a feature of k-means clustering which is that the centroids will end up in the same resting place (or very close to it) regardless of where they started. The only caveat related to centroid positioning is that no two centroids should occupy the same initial location.

Now that we have our centroids, they are now subjected to a 2-step iterative re-location process. For each iteration we determine which entities are closest to which centroids, then we recalculate the position of the centroids based as the mean location of all of the entities affiliated with them.

To set up this process, we make the following call to the KMeans routine:

```
MyKMeans:=ML.Cluster.KMeans(dEntities,dCentroids,30,.3);
```

Here, we are passing in our two datasets, dEntities and dCentroids. In order to prevent infinite loops, we also must specify a maximum number of iterations, which is set to 30 in the above example.

Convergence is defined as the point at which we can say the centroids have found their final resting places. Ideally, this will be when they stop moving completely.

However, there will be situations where centroids may experience a "see-sawing" action, constantly trading affiliations back and forth indefinitely. To address this, we have the option of specifying a positive value as the convergence threshold. The process will assume convergence if, during any iteration, no centroid moves a distance greater than that number. In our above example, we are setting the convergence threshold to 0.3. If no threshold is specified, then the threshold is set to 0.0. If the process hits the maximum number of iterations passed in as parameter 3, then it stops regardless of whether convergence is achieved or not.

The final parameter, which is also optional, specifies which distance formula to use. For our example we are leaving this parameter blank, so it defaults to a simple Euclidean calculation, but we could easily change this by adding the fifth parameter with a value such as "ML.Cluster.DF.Tanimoto" or "ML.Cluster.DF.Manhattan".

Below are calls to the available attributes within the KMeans module:

```
MyKMeans.AllResults;
```

This will produce a table with a layout similar to NumericField, but instead of a single value field, we have a field named "values" which is a set of values.

Each row will have the same number of values in this set, which is equal to the number of iterations + 1. Values[1] is the initial value for the id/number combination, Values[2] is after the first iteration, etc.

```
MyKMeans.Convergence;
```

Convergence will respond with the number of iterations that were performed, which will be an integer between 1 and the maximum specified in the parameters. If it is equal to the maximum, then you may want to increase that number or specify a higher convergence threshold because it had not yet achieved convergence when it completed.

```
MyKMeans.Result();  // The final locations of the centroids
MyKMeans.Result(3); // The results of iteration 3
```

Results will respond with the centroid locations after the specified number of iterations. If no number is passed, this will be the locations after the final iteration.

```
MyKMeans.Delta(3,5); // The distance every centroid travelled across each axis from
                        iterations 3 to 5
MyKMeans.Delta(0);   // The total distance the centroids travelled on each axis from the
                        beginning to the end
```

Delta displays the distance traveled *on each axis* between the iterations specified in the parameters. If no parameters are passed, this will be the delta between the last two iterations.

```
MyKMeans.DistanceDelta(3,5); // The straight-line distance travelled by each centroid from
                                iterations 3 to 5
MyKMeans.DistanceDelta(0);   // The total straight-line distance each centroid travelled
MyKMeans.DistanceDelta();    // The distance traveled by each centroid during the last
                                iteration.
```

DistanceDelta is the same as Delta, but displays the DISTANCE delta as calculated using whichever method the KMeans routine was instructed to use, which in our example is Euclidean.

The function Allegiances provides a table of all the entities and the centroids to which they are closest, along with the actual distance between them. If a parameter is passed, it is the iteration number after which to sample the allegiances. If no parameter is passed, the convergence iteration is assumed.

A second function, Allegiance, enables the user to pinpoint a specific entity to determine its allegiance. This function requires one parameter, which is the ID of the entity to poll. The second parameter is the iteration number and has the same behavior as with Allegiances. The return value for Allegiance is an integer representing the value of the centroid to which the entity is allied.

```
MyKMeans.Allegiances();     // The table of allegiances after convergence
MyKMeans.Allegiance(10,5);  // The centroid to which entity #10 is closest after iteration 5
```

AggloN

With Agglomerative, or Hierarchical, clustering there is no need for a centroid set. This method takes a bottom-up approach whereby it identifies those pairs that are mutually closest and marries them so they are treated as a single entity during the next iteration. Allowed to run until full convergence, every entity will eventually be stitched up into a single tree structure with each fork representing tighter and tighter clusters.

We set up this clustering routine using the following call:

```
MyAggloN:=ML.Cluster.AggloN(dEntities,4);
```

Here, we are passing in our sample data set and telling the routine that we want a maximum of 4 iterations.

There are two further parameters that the user may pass, both of which are optional. Parameter 3 enables the user to specify the distance formula exactly as we could in Parameter 5 of the KMeans routine. And as with our KMeans example, we will leave this blank so it defaults to Euclidean.

Parameter 4 enables us to specify how we want to represent distances where clustered entities are involved. After the first iteration some of the entities will have been grouped together and we need to make a decision about how we measure distance to those groups. The three options are min_dist, max_dist, and ave_dist, which will instruct the routine to use the minimum distance within the cluster, the maximum or the average respectively. The default, which we are accepting for this example, is min_dist.

The following three calls will give us the results of the Agglomerative clustering call in different ways:

```
MyAggloN.Dendrogram;
```

The Dendrogram call displays the output as a string representation of the tree diagram. Clusters are grouped within curly braces ({}), and clusters of clusters are grouped in the same manner. The ID for each cluster will be assigned the lowest ID of the entities it encompasses. In our example, we end up with five clusters, and two entities yet to be clustered. This is because we specified a maximum of four iterations which was not enough to group everything together.

```
MyAggloN.Distances;
```

The Distances output displays all of the remaining distances that would be used to further cluster the entities. If we had achieved convergence, this would be an empty table and our Dendrogram output would be a single line with every item found within the tree string. But since we stopped iterating early, we still have items to cluster, and therefore still have distances to display. The number of rows here will be equal to n*n-1, where n is the number of rows in the Dendrogram table.

```
MyAggloN.Clusters;
```

Clusters will display each entity, and the ID of the cluster that the entity was assigned to. In our example, every entity will be assigned to one of the seven cluster IDs found in the Dendrogram. If we had allowed the process to continue to convergence, which for our sample set is achieved after 9 iterations, every entity will be assigned the same cluster ID because it will be the only one left in the Dendrogram.

Correlations Walk-through

Most of the algorithms within the ML libraries assume that one of your inputs are a collection features of a particular object and the algorithm exists to predict some other feature based upon the features you have. In the literature the ‘features you have’ are usually referred to as the ‘independent variables’ and the features you are trying to predict are called the ‘dependent variables’.

Masked within those names is an assumption that is almost never true. That the features you have for a given object are actually independent of each other. Consider, for example, a classification algorithm that tries to predict risk of heart disease based upon height, weight, age and gender. The independent variables are not even close to being independent. Pick any two of those variables and there is a known link between them (even age and gender; women live longer). These linkages between the ‘independent’ variables usually represent an error factor in the algorithm used to compute the dependent variable.

The Correlation module exists to allow you to quantify the degree of relatedness between a set of variables. There are three measures provided. Under the title ‘simple’ the Covariance and Pearson statistic is provided for every pair of variables. The Kendall measure provides the Kendal Tau statistic for every pair of variables; it should be noted that computation of Kendall’s Tau is an O(N^2) process. This will hurt on very large datasets.

The definition and interpretation of these terms can be found in any statistical text; for example: http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient http://en.wikipedia.org/wiki/Kendall_tau_rank_correlation_coefficient

(we implement tau-a)

```
import ml;

value_record := RECORD
                unsigned rid;
  real height;
                real weight;
                real age;
                integer1 species;
                integer1 gender; // 0 = unknown, 1 = male, 2 = female
  END;

d := dataset([{1,5*12+7,156*16,43,1,1},
              {2,5*12+7,128*16,31,1,2},
              {3,5*12+9,135*16,15,1,1},
              {4,5*12+7,145*16,14,1,1},
              {5,5*12-2,80*16,9,1,1},
              {6,4*12+8,72*16,8,1,1},
              {7,8,32,2.5,2,2},
              {8,6.5,28,2,2,2},
              {9,6.5,28,2,2,2},
              {10,6.5,21,2,2,1},
              {11,4,15,1,2,0},
              {12,3,10.5,1,2,0},
              {13,2.5,3,0.8,2,0},
              {14,1,1,0.4,2,0}
             ]
             ,value_record);
// Turn into regular NumericField file (with continuous variables)
ml.ToField(d,o);

Cor := ML.Correlate(o);
Cor.Simple;
Cor.Kendall;
```

Discretize Walk-through

Modules: Discretize

As discussed briefly in the section on data models, it is not unusual for data to be provided in a manner where the data forms some real value. For example a height or weight might be measured down to the inch or ounce, or a price might be measured down to the nearest cent. Yet in terms of predictiveness we might expect *similar* values in those fields to exhibit similar behavior in some particular regard.

Some of the ML modules expect the input data to have been *banded*, or for data which was originally in *real values* to have been turn into a set of discrete bands. More concretely, they require data in the DiscreteField format even if it was originally provided in NumericField format.

The Discretize module exists to perform this conversion. All of the examples in this walk-through use the first dataset (d) from the NumericField walk-through. It might help you to just quickly look at that dataset again in both original and NumericField form to remind you of the format.

```
ml.ToField(d,o);
d;
o;
```

There are currently 3 main methods available to create discrete values, ByRounding, ByBucketing and ByTiling.

All three methods operate upon all the data handed to them. Applying different methods to different fields is very easy using the methods discussed in the section on Data Models. This simple example Auto-buckets columns 2 & 3 into four bands, tiles column 1 into 6 bands and the rounds the fourth column to the nearest integer:

```
disc := ML.Discretize.ByBucketing(o(Number IN [2,3]),4)+ML.Discretize.ByTiling(o(Number IN
                                              [1]),6)+ML.Discretize.ByRounding(o(Number=4));
disc;
```

It may be observed that in the description above I was able to describe how to discretize data a number of different ways but could not give any firm guidelines as to the exactly number of bands or the exact best method to use for any given item of data. That is because whilst there are schemes and guidelines out there, there is no firm consensus as to which are the best. It is quite possible that the best way to work is to ‘try a few’ and see which gives you the best results!

The Discretize module supports this ‘suck it and see’ approach by allowing you to specify the discretization methods entirely within data. The core of this is the ‘Do’ command that takes a series of instructions and discretizes a dataset based upon those instructions. Each instruction is actually a little record of type r_Method defined in the Discretize module and you can construct these records yourself if you wish. It would be fairly easy to even create a little *discreting programming language*’ and have it execute. For the slightly less ambitious there are a collection of parameterized functions that will construct an r_Method record for each of the three main discretize types.

The following example is exactly equivalent to the previous:

```
// Build up the instructions into a single data file (‘mini program’)

inst := ML.Discretize.i_ByBucketing([2,3],4)+ML.Discretize.i_ByTiling([1],6)
                                            +ML.Discretize.i_ByRounding([4]);

// Execute the instructions

done := ML.Discretize.Do(o,inst);
done;
```

ByRounding

The ByRounding method exists to convert a real number to an integer by ‘rounding it’. At its simplest this means that every real number is converted to an integer. Values that were less than a half go down to the nearest integer; those that were .5 or above go up. Therefore if you have a field that has house prices; perhaps from $10,000 to $1M then you potentially will end up with 990,000 different discrete values (ever possible dollar value).

This has made the data discrete but it hasn’t really satisfied the problem that ‘similar values’ have an identical discrete value. We might expect a $299,999 dollar house to be quite similar to a $299,998 dollar house. The ByRounding method therefore has a scale. The real number in the data is multiplied by the scale factor PRIOR to rounding.

In our example, if we apply a scale of 0.0001 (1/10000) a $299999 house (and a $299998 house) will both get a ByRounding result of 30. The scale effectively reduces the range of a variable. In the house case a scale of 0.0001 reduces the range from “$10,000 to $1M” to 1-100 which is much more manageable.

Sometimes the scaled ranges do not work out so neatly. Suppose the field is measuring height of high-school seniors. The original range is probably from 48 inches up to possibly 88 inches. A scale of 0.25 is probably enough to give the number of discrete values you require (10), but they will range from 12 to 22 which is not convenient. Therefore a DELTA is available, which is ADDED to the value AFTER scaling but before rounding. It can therefore be used to bring the eventual range down to a convenient number. In this case a Delta of -11 would give us an eventual range of 1-11, which is perfect.

ByBucketing

ByBucketing is mathematically similar to ByRounding but with rather more ease of use. There is a slight performance hit and rather less control with the ByBucketing method. Within the ByBucketing method you do not specify the scale or the delta, you simply specify the number of eventual buckets (or the number of discrete values) that you eventually want. It does a pre-pass of the data to compute the scale and delta before applying it.

ByTiling

Both of the previous methods divide the banding evenly across the range of the original variable. However, while the range has been divided evenly, the number of different records within each band could vary greatly. In the height example, one would expect a large number of children within the 60-72 inch range (rounded values of 3-7) but very few in bands 1 or 11.

An alternative approach is not to band by some absolute range but rather to band on the value of a given value relative to all of the other records. For example, you may want to end with 10 bands where each band has the same number of records in it; and band 10 is the top 10% of the population, band nine is the second 10% etc. This result is achieved using the ByTiling scheme. Similar to ByBucketing you specify the number of bands you eventually want and the system will automatically allocate the field values for you.

Note: ByTiling does require the data to be sorted and so will have an NLgN performance profile.

Docs Walk-through

Modules: Docs

The processing of textual data is a unique problem in the field of Machine Learning because of the highly unstructured manner in which humans write. There are many ways to say the same thing and many ways that different statements can look alike. There is an enormous body of research that has been performed to determine algorithms for accurately and efficiently extracting useful information from data such as electronic documents, articles, and transcriptions, all of which rely on human speech patterns.

The Docs module of the ML library is designed to help prepare unstructured and semi-structured text to make it more suitable for further processing. This includes routines to decompose the text into discrete word elements and collating simple statistics on those tokens, such as Term Frequency and Inverse Document Frequency. Also included are the basic tools to help determine token association strength using industry-standard functions such as Support and Confidence.

Tokenize

The Tokenize module breaks a set of raw text into its lexical elements. From there, it can produce a dictionary of those elements with weighting as well as perform integer replacement that significantly reduces the space overhead needed to process such large amounts of data.

For the purposes of this walk-through we will be using the following limited dataset:

```
IMPORT ML;

dSentences:=DATASET([

  {1,'David went to the market and bought milk and bread'},
  {2,'John picked up butter on his way home from work.'},
  {3,'Jill craved lemon cookies, so she grabbed some at the convenience store'},
  {4,'Mary needs milk, bread and butter to make breakfast tomorrow morning.'},
  {5,'William\'s lunch included a sandwich on wheat bread and chocolate chip cookies.'}

],ML.Docs.Types.Raw);
```

The format of the initial dataset is in the Raw format in Docs.Types, which is a simple numeric ID and a string of free text of indeterminate length.

It is important that a unique ID is assigned to each row so that we can have references not just for every word, but for every document as well.

In the above dataset we already have assigned these IDs, but if your input table does not yet have them, a quick call to Tokenize.Enumerate will assign a sequential integer ID to the table:

```
dSequenced:=ML.Docs.Tokenize.Enumerate(dSentences)
```

The first step in parsing the text is to run it through the Clean function. This is a simple function that standardizes the text by performing actions such as removing punctuation, converting all letters into capitals, and normalizing some common contractions.

```
dCleaned:=ML.Docs.Tokenize.Clean(dSentences);
```

Once cleaned, the next step is to break out each word as a separate entity using the Split function. A word is defined intuitively as a series of non-white-space characters surrounded by white space.

```
dSplit:=ML.Docs.Tokenize.Split(dCleaned);
```

The output produced from the Split function is a 3-column table in ML.Docs.Types.WordElement format, with the document ID, the ordinal position of the word within the text of that document, and the word itself.

In our example, the first few rows of this table will be:

```
1   1   DAVID
1   2   WENT
1   3   TO
```

This opens us up a number of possibilities for processing our text. Most, require one further step, which is to derive some aggregate information of the words that appear in our corpus of documents. We do this using the Lexicon function:

```
dLexicon:=ML.Docs.Tokenize.Lexicon(dSplit)
```

This function aggregates the data in our dSplit table, grouping on word. The resulting dataset contains one row for each word along with a unique ID (an integer starting at 1), a total count of the number of times the word occurs in the entire corpus, and the number of unique documents within which the word exists. The ID assigned to the word is inversely proportional to the word frequency, which means that the word that appears the most often will be assigned 1, the next most common will have 2, and so on.

When processing very large amounts of text, there is an additional function ToO which can be used to reduce the amount of resources used during processing:

```
dReplaced:=ML.Docs.Tokenize.ToO(dSplit,dLexicon);
```

The output from this function will have as many rows as there are in dSplit, but instead of seeing the words as they are in the text, you will see the word ID that was assigned to it in dLexicon. This saves a large amount of memory because the word ID is always 4-byte integer, while the word is variable length and usually much larger. Since the function has access to the aggregate information collected by the Lexicon function, this information is also tacked back on to the output from ToO so that it is readily available if desired.

From this point, we have the framework for performing numerous Natural Language Processing algorithms, such as keyword designation and extraction using the TF/IDF method, or even clustering by treating each word ID as a dimension in Euclidean space.

Finally, the function FromO is pretty self-explanatory. This simply re-constitutes a table that was produced by the ToO function back into the WordElement format.

```
dReconstituted:=ML.Docs.Tokenize.FromO(dReplaced,dLexicon);
```

Co-Location

The Docs.CoLocation module takes the textual analysis one step further than Tokenize. It harvests n-grams rather than just single words and enables the user to perform analyses on those n-grams to determine significance. The same dataset (dSentences) that was used in the walk-through of the Tokenize module above, is also used as the starting point for the examples shown below. As with Tokenize, the first step in processing the free text for Colocation is to map all of the words. This is done by calling the Words attribute, which also calls the Tokenize.Clean and Tokenize.Split functions respectively:

```
dWords:=ML.Docs.CoLocation.Words(dSentences);
```

The AllNGrams attribute then harvests every n-gram, from unigrams up to the n defined by the user. This produces result in a table that contains a row for every unique id/n-gram combination. In the following line, we are asking for anything up to a 4-gram. If the n parameter is left blank, the default is 3.

```
dAllNGrams:=ML.Docs.CoLocation.AllNGrams(dWords,,4);
```

Note: The above call has left the second parameter blank.

The second parameter is a reference to a Lexicon which is used if you decide to perform integer replacement on the words prior to processing. This is advisable for very large corpuses. In such a case, we would have first called the Lexicon function (which exists in CoLocation as a pass-through of the same function in Tokenize) and is then passed that output as the second parameter:

```
dLexicon:=ML.Docs.CoLocation.Lexicon(dWords);
dAllNGrams:=ML.Docs.CoLocation.AllNGrams(dWords,dLexicon,4);
```

Below are calls to the standard metrics that are currently built into the CoLocation module. Remember that the call to Words above has called Tokenize.Clean, which has converted all characters in the text to uppercase:

```
// SUPPORT: User passes a SET OF STRING and the output from the ALLNGrams attribute
ML.Docs.CoLocation.Support(['MILK','BREAD','BUTTER'],dAllNGrams);

// CONFIDENCE, LIFT and CONVICTION: User passes in two SETS OF STRING and the AllNGrams
   output.
// In each case, set 1 and set 2 are read as “1=>2”.  Note that 1=>2 DOES NOT EQUAL 2=>1.
ML.Docs.CoLocation.Confidence(['MILK','BREAD'],['BUTTER'],dAllNGrams);
ML.Docs.CoLocation.Lift(['MILK','BREAD'],['BUTTER'],dAllNGrams);
ML.Docs.CoLocation.Conviction(['MILK','BREAD'],['BUTTER'],dAllNGrams);
```

To further distill the data the user may call NGrams. This strips the document IDs and groups the table so that there is one row per unique n-gram. Included in this output is aggregate information including the number of documents in which the item appears, the percentage of that compared to the document count, and the Inverse Document Frequency (IDF).

```
dNGrams:=Docs.CoLocation.NGrams(dAllNGrams);
```

With the output from NGrams there are other attributes that can be called to further analyze the data.

Calling SubGrams produces a table of every n-gram where n>1 along with a comparison of the document frequency of the n-gram to the product of the frequencies of all of its constituent unigrams.

This gives an indication of whether the phrase or its parts may be more significant in the context of the corpus.

```
ML.Docs.CoLocation.SubGrams(dNGrams);
```

Another measure of significance is SplitCompare. This splits every n-gram with n>1 into two rows with two parts which are the initial unigram and the remainder, and the final unigram and the remainder. The document frequencies of all three items (the full n-gram, and the two constituent parts) are then presented side-by-side so their relative values can be evaluated. This helps to determine if a leading or trailing word carries any weight in the encompassing phrase.

```
ML.Docs.CoLocation.SplitCompare(dNGrams);
```

Once any analysis has been done and the user has phrases of significance, they can be re-constituted using a call to ShowPhrase:

```
ML.Docs.CoLocation.ShowPhrase(dLexicon,’14 13 4’);  // would return ‘CHOCOLATE CHIP COOKIES’
```

Field Aggregates Walkthrough

Modules: FieldAggregates, Distribution

The FieldAggregates module exists to provide statistics upon each of the fields of a file. The file is passed in to the field aggregates module and then various properties of those fields can be queried, for example:

```
IMPORT ML;
// Generate random data for testing purposes
TestSize := 10000000;
a1 := ML.Distribution.Uniform(0,100,10000);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
a2 := ML.Distribution.Poisson(3,100);
b2 := ML.Distribution.GenData(TestSize,a2,2);
D := b1+b2; // This is the test data
// Pass the test data into the Aggregate Module
Agg := ML.FieldAggregates(D);
Agg.Simple; // Compute some common statistics
```

This example provides two rows. The ‘number’ column ties the result back to the column being passed in. There are columns for min-value, max-value, the sum, the number of rows (with values), the mean, the variance and the standard deviation. The ‘simple’ attribute is a very good one to use on huge data as it is a simple linear process.

The aggregate module is also able to ‘rank order’ a set of data; the SimpleRanked attribute allocates every value in every field a number – the smallest value gets the number 1, then 2 etc. The ‘Simple’ indicator is to denote that if a value is repeated the attribute will just arbitrarily pick which one gets the lower ranking.

As you might expect there is also a ‘ranked’ attribute. In the case of multiple identical values this will assign every value with the same value a rank which is the average value of the ranks of the individual items, for example:

```
IMPORT ML;
TestSize := 50;
a1 := ML.Distribution.Uniform(0,100,10000);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
a2 := ML.Distribution.Poisson(3,100);
b2 := ML.Distribution.GenData(TestSize,a2,2);
D := b1+b2; // This is the test data
Agg := ML.FieldAggregates(D);
Agg.SimpleRanked;
Agg.Ranked;
```

Note: Ranking requires the data to be sorted; therefore ranking is an ‘NlgN’ process.

When examining the results of the ‘Simple’ attribute you may be surprised that two of the common averages ‘median’ and ‘mode’ are missing. While the Aggregate module can return those values, they are not included in the ‘Simple’ attribute because they are NLgN processes and we want to keep ‘Simple’ as cheap as possible. The median values for each column can be obtained using the following:

```
Agg.Medians;
```

The modes are found by using:

```
Agg.Modes;
```

It is possible that more than one mode will be returned for a particular column, if more than one value has an equal count.

The final group of features provided by the Aggregate module are the NTiles and the Buckets. These are closely related but totally different which can be confusing.

The NTiles are closely related to terms like ‘percentiles’, ‘deciles’ and ‘quartiles’, which allow you to grade each score according the a ‘percentile’ of the population. The name ‘N’ tile is there because you get to pick the number of groups the population is split into. Use NTile(4) for quartiles, NTile(10) for deciles and NTile(100) for percentiles. NTile(1000) can be used if you want to be able to split populations to one tenth of a percent. Every group (or Tile) will have the same number of records within it (unless your data has a lot of duplicate values because identical values land in the same tile). The following example demonstrates the possible use of NTiling.

Imagine you have a file with people and for each person you have two columns (height and weight). NTile that file with a number, such as 100. Then if the NTile of the Weight is much higher than the NTile of the Height, the person might be overweight. Conversely if the NTile of the Height is much higher than the Weight then the person might be underweight. If the two percentiles are the same then the person is ‘normal’.

NTileRanges returns information about the highest and lowest value in every Tile. Suppose you want to answer the question: “what are the normal SAT scores for someone going to this college”. You can compute the NTileRanges(4). Then you can note both the low value of the second quartiles and the high value of the third quartile and declare that “the middle 50% of the students attending that college score between X and Y”.

The following example demonstrates this:

```
IMPORT ML;
TestSize := 100;
a1 := ML.Distribution.Uniform(0,100,10000);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
a2 := ML.Distribution.Poisson(3,100);
b2 := ML.Distribution.GenData(TestSize,a2,2);
D := b1+b2; // This is the test data
Agg := ML.FieldAggregates(D);
Agg.NTiles(4);
Agg.NTileRanges(4)
```

Buckets provide very similar looking results. However buckets do NOT attempt to divide the groups so that the population of each group is even. Buckets are divided so that the RANGE of each group is even. Suppose that you have a field with a MIN of 0 and MAX of 50 and you ask for 10 buckets, the first bucket will be 0 to (almost)5, the second 5 to (almost) 10 etc. The Buckets attribute assigns each field value to the bucket. The BucketRanges returns a table showing the range of each bucket and also the number of elements in that bucket. If you wanted to plot a histogram of value verses frequency, for example, buckets would be the tool to use.

The final point to mention is that many of the more sophisticated measures use the simpler measures and also share other more complex code between themselves. If you eventually want two or more of these measures for the same data it is better to compute them all at once. The ECL optimizer does an excellent job of making sure code is only executed once however often it is used. If you are familiar with ECL at a lower level, you may wish to look at the graph for the following:

```
IMPORT ML;
TestSize := 10000000;
a1 := ML.Distribution.Uniform(0,100,10000);
b1 := ML.Distribution.GenData(TestSize,a1,1); // Field 1 Uniform
a2 := ML.Distribution.Poisson(3,100);
b2 := ML.Distribution.GenData(TestSize,a2,2);
D := b1+b2; // This is the test data
Agg := ML.FieldAggregates(D);
Agg.Simple;
Agg.SimpleRanked;
Agg.Ranked;
Agg.Modes;
Agg.Medians;
Agg.NTiles(4);
Agg.NTileRanges(4);
Agg.Buckets(4);
Agg.BucketRanges(4)
```

Matrix Library Walk-through

The Matrix Library provides a number of matrix manipulation routines. Some of them are standard matrix operations that do not require any specific explanations (Add, Det, Inv, Mul, Scale, Sub, and Trans). Others are a bit less standard, and have been created to provide the appropriate functional support for other ML library algorithms.

```
IMPORT ML;
IMPORT ML.Mat AS Mat;
d := dataset([{1,1,1.0},{1,2,2.0},{2,1,3.0},{2,2,4.0}],Mat.Types.Element);
d1:= Mat.Scale(d,10.0);
Mat.Add(d1,d);
Mat.Sub(d1, d );
Mat.Mul(d,d);
Mat.Trans(d);
Mat.Inv(d);
```

Each

The Each matrix module provides routines for element-wise matrix, in that it provides functions that operate on individual elements of the matrix. The following code starts with the square matrix whose elements are equal to 2.

```
IMPORT * FROM ML;
A := dataset([{1,1,2.0},{1,2,2.0},{1,3,2.0},
              {2,1,2.0}, {2,2,2.0},{2,3,2.0},
              {3,1,2.0},{3,2,2.0}, {3,3,2.0}], ML.Mat.Types.Element);
AA := ML.Mat.Each.Mul(A,A);
A_org := ML.Mat.Each.Sqrt(AA);
OneOverA := ML.Mat.Each.Reciprocal(A_org,1);
ML.Mat.Each.Mul(A_org,OneOverA);
```

The Each.Mul routine multiplies each element of the matrix A with itself producing the matrix AA whose elements are equal to 4. The Each.Sqrt routine calculates the square root of each element producing the matrix A_org whose elements are equal to 2. The Each.Reciprocal routine calculates reciprocal value of every element of the matrix A_org producing the matrix OneOverA whose elements are equal to ½.

Has

The Has matrix module provides various matrix properties, such as matrix dimension or matrix density.

Is

The Is matrix module provides routines to test matrix types, such as whether a matrix is an identity matrix, or whether it is a zero matrix, or a diagonal matrix, or a symmetric matrix, or a Upper or Lower triangular matrix.

Insert Column

You may need to insert a new column into an existing matrix, e.g. regression analysis usually requires a column of 1s to be inserted into the feature matrix X before a regression model gets created. InsertColumn was created for this purpose. The following inserts a column of 1s as the first column into the square matrix A, creating the 3-by-4 matrix:

```
IMPORT * FROM ML;
A := dataset([{1,1,2.0},{1,2,3.0},{1,3,4.0},
              {2,1,2.0}, {2,2,3.0},{2,3,4.0},
              {3,1,2.0},{3,2,3.0}, {3,3,4.0}], ML.Mat.Types.Element);
ML.Mat.InsertColumn(A, 1, 1.0);
```

MU

MU is a matrix universe module. Its routines make it possible to include multiple matrices into the same file. These routines are useful when it is necessary to return more than one matrix from a function. For example, the QR matrix decomposition process produces 2 matrices, Q and R, and those two matrices can be combined together using routines from the MU module.

This sample code starts with 2 square 3-by-3 matrices, A1 and A2. One with all elements eqaal to 1 and the other with all elements equal to 2. The 2 matrices are combined into one universal matrix A1MU + A2MU, with id=4 identifying elements of the matrix A1 and id=7 identifying elements of matrix A2. The last two code lines extract the original matrices from the universal matrix A1MU + A2MU.

```
IMPORT * FROM ML;

A1 := dataset([{1,1,1.0},{1,2,1.0},{1,3,1.0},
                         {2,1,1.0}, {2,2,1.0},{2,3,1.0},
                         {3,1,1.0},{3,2,1.0}, {3,3,1.0}], ML.Mat.Types.Element);

A2 := dataset([{1,1,2.0},{1,2,2.0},{1,3,2.0},
                         {2,1,2.0}, {2,2,2.0},{2,3,2.0},
                         {3,1,2.0},{3,2,2.0}, {3,3,2.0}], ML.Mat.Types.Element);

A1MU := ML.Mat.MU.To(A1, 4);
A2MU := ML.Mat.MU.To(A2, 7);
A1MU+A2MU;
ML.Mat.MU.From(A1MU+A2MU, 4);
ML.Mat.MU.From(A1MU+A2MU, 7);
```

Repmat

The Repmat function replicates a matrix, creating a large matrix consisting of M-by-N tiling copies of the original matrix. For example, the following code starts from a matrix with one element with value = 2. It then creates a 3x2 matrix out of it by replicating this single element matrix 3 times vertically, to create a 3x1 vector. This vector is then replicated 2 times horizontally.

```
IMPORT * FROM ML;

A := DATASET ([{1,1,2.0}], ML.Mat.Types.Element);
B := ML.Mat.Repmat(A,3,2);
```

The resulting matrix B is a 3x2 matrix with all elements having a value of 2, as in:

```
DATASET([{1,1,2.0},{1,2,2.0},{2,1,2.0}, {2,2,2.0},{3,1,2.0},{3,2,2.0}], ML.Mat.Types.Element);
```

The Repmat function can be used to adjust the mean values of the columns of a given matrix. This can be achieved by first calculating the mean values of every matrix column using the mcA := Has(A).MeanCol. This function generates a row vector mcA containing mean values for every matrix column. This row vector then needs to be replicated vertically to match the size of the original matrix, which can be achieved using the rmcA := Repmat(mcA, Has(A).Stats.XMax, 1). The rmcA matrix is the same size as the original matrix A, and its column values are the same for every matrix column and they are equal to the mean value of that column. Finally, if we subtract the rmcA from matrix A, we get a matrix whose columns have the mean value of zero. This can be achieved using the following compact code:

```
IMPORT * FROM ML;
A := dataset([{1,1,2.0},{1,2,3.0},{1,3,4.0},
              {2,1,2.0}, {2,2,3.0},{2,3,4.0},
              {3,1,2.0},{3,2,3.0}, {3,3,4.0}], ML.Mat.Types.Element);
ZeroMeanA := ML.Mat.Sub(A, ML.Mat.Repmat(ML.Mat.Has(A).MeanCol,
ML.Mat.Has(A).Stats.XMax, 1));
```

Decomp

The Decomp matrix module provides routines for different matrix decompositions (or matrix factorizations). Different decompositions are needed to implement efficient matrix algorithms for particular cases of problems in linear algebra.

LU Decomposition

The LU matrix decomposition is applicable to a square matrix A, and it is used to help solve a system of linear equations Ax = b. When solving a system of linear equations Ax = b, the matrix A can be decomposed via the LU decomposition, which factorizes a matrix into a lower triangular matrix L and an upper triangular matrix U. The equivalent systems L(Ux) = b and Ux = Inv(L)b are easier to solve then the original system of linear equations Ax = b. These equivalent systems of linear equations are solved by 'forward substitution' and 'back substitution' using the f_sub and b_sub routines available in the Decomp module. The LU decomposition is currently being used to calculate the inverted matrix.

The following code demonstrates how to decompose matrix A into its L and U components. The L and U components are calculated first. To validate that this matrix decomposition is done correctly, we need to demonstrate that A=LU. This code does that by multiplying L and U, and then subtracting that result from A. The expected result is a zero matrix (the matrix whose size is the same as the size of the original matrix A with all elements being equal to 0).

The problem is that the arithmetic involved in calculation of L and U components may create some rounding error, and as a result of that the A-LU matrix may not have all zero elements, as some elements could be positive real numbers very close to zero. For example, the element (4,3) of the matrix A-LU in the following example has the value 8.881784197001252e-16, the number that differs from 0 at its 15th decimal point. The sample code deals with that problem by applying the RoundDelta function to the A-LU matrix.

```
IMPORT * FROM ML;
A := dataset([{1,1,2.0},{1,2,-1.0},{1,3,3.0},{1,4,4.0},
              {2,1,4.0}, {2,2,2.0},{2,3,1.0},{2,4,5.0},
              {3,1,-6.0},{3,2,-1.0}, {3,3,3.0}, {3,4,6.0},
              {4,1,-6.0},{4,2,-1.0}, {4,3,1.0}, {4,4,6.0}], ML.Mat.Types.Element);
L := ML.Mat.Decomp.LComp(ML.Mat.Decomp.LU(A));
U := Ml.Mat.Decomp.UComp(ML.Mat.Decomp.LU(A));
LU := ML.Mat.Mul(L,U);
ML.MAT.RoundDelta(ML.Mat.Sub(A,LU));
```

Cholesky Decomposition

The Cholesky matrix decomposition is applicable to a square, symmetric, and positive real matrix A. It factorizes a matrix A into the product of a lower triangular matrix L, and its transpose L'. It is mainly used for the numerical solution of system of linear equations Ax = b. If A is symmetric and positive, then a system of linear equations Ax = b can be solved by first computing the Cholesky decomposition A = L*L', then solving Ly = b for y, and finally solving L'x = y for x.

The sample code demonstrates how to create a lower triangular matrix L using Cholesky decomposition. The code validates this decomposition by multiplying L with its transpose and ensuring that this product is equal to the original matrix A.

```
IMPORT * FROM ML;
A := dataset([{1,1,2.0},{1,2,1.0},{1,3,1.0},
              {2,1,1.0}, {2,2,3.0},{2,3,1.0},
             {3,1,1.0},{3,2,1.0}, {3,3,4.0}], ML.Mat.Types.Element);
 L := ML.Mat.Decomp.Cholesky(A);
LLt := ML.Mat.Mul(L,ML.Mat.Trans(L));
ML.MAT.RoundDelta(ML.Mat.Sub(A,LLt));
```

QR Decomposition

The QR matrix decomposition is applicable to an m-by-n matrix A. It factorizes matrix A into a product QR, where Q is an orthogonal matrix (ie Q*Q’ = I) of size m-by-m, and R is an upper triangular matrix of size m-by-n. The QR decomposition provides an alternative way of solving a system of linear equations Ax = b, without having to invert the matrix. Since the Q is orthonormal, the original system Ax = b is equivalent to Rx = Q’b = c. The system Rx = c is solved by ‘back substitution’, since R is triangular matrix.

The following sample code demonstrates how to decompose matrix A into its Q and R components. The code validates this decomposition by demonstrating that a/ A=QR, and b/ Q is an orthonormal matrix. The purpose of the Thin() function is to create a sparse matrix representation of its input matrix, and this is achieved by removing all zero elements from it. Since the A-QR is expected to be a zero matrix, applying the Thin() function to it is expected to create an empty matrix. The second result is expected to be an identity matrix, and the Thin() function will remove all zero elements from it.

```
IMPORT * FROM ML;
A := dataset([{1,1,12.0},{2,1,6.0},{3,1,-4.0},
              {1,2,-51.0},{2,2,167.0},{3,2,24.0},
              {1,3,4.0},{2,3,-68.0},{3,3,-41.0}], ML.MAT.Types.Element);

Q := ML.Mat.Decomp.QComp(A);
R := ML.Mat.Decomp.RComp(A);
QR := ML.Mat.Mul(Q,R);
ML.Mat.Thin(ML.MAT.RoundDelta(ML.Mat.Sub(A,QR)));
ML.Mat.Thin(ML.MAT.RoundDelta(ML.Mat.Mul(Q,ML.Mat.Trans(Q))));
```

Eigen Decomposition (Eigenvalues and Eigenvectors)

Eigen decomposition is a factorization of a matrix, where the original matrix is represented in terms of its eigenvalues and eigenvectors. Indirect definitions of eigenvalues and eigenvectors can be stated as, "given a square matrix A, the number λ is an eigenvalue of A if there exists a non-zero vector v such that A*v =λ*v." If such a vector v exists, then v is called an eigenvector of A corresponding to the eigenvalue λ.

In general, when matrix is multiplied with a vector, it changes both the vector’s magnitude and direction. However, for some vectors, A*v can only change the vector’s magnitude, while leaving its direction unchanged. These vectors are eigenvectors of the matrix. A matrix changes a vector’s magnitude by multiplying its magnitude by a factor, which is positive if the vector’s direction does not change and is negative if the vector’s direction is reversed. This factor is the eigenvalue λ associated with that eigenvector v.

Here is another way of looking at eigenvectors and eigenvalues. An eigenvector of matrix A is a vector that maintains its direction after undergoing a linear transformation A (i.e. after matrix multiplication A*v), and an eigenvalue is the scalar value that the eigenvector was multiplied by during the linear transformation A.

Eigen decomposition can be expressed as matrix factorization into matrices V and L which satisfy the following equation: A*V = V*L, where V is the square matrix whose columns are the eigenvectors of A, and L is a diagonal matrix whose diagonal elements are eigenvalues of A. The code below demonstrates how to use the Eig function to perform an Eigen decomposition of a matrix:

```
IMPORT * FROM ML;
A := dataset([{1,1,12.0},{2,1,6.0},{3,1,4.0},
              {1,2,6.0},{2,2,167.0},{3,2,24.0},
              {1,3,4.0},{2,3,24.0},{3,3,-41.0}], ML.MAT.Types.Element);

eig_values:=ML.Mat.Eig(A).valuesM;
eig_valuesV:=ML.Mat.Eig(A).valuesV;
eig_vectors:=ML.Mat.Eig(A).vectors;
cnt:=(INTEGER)ML.Mat.Eig(A).convergence;
```

The ValuesM attribute returns eigenvalues in the diagonal matrix format. The valueV returns those very same eigenvalues but in a vector format. The vectors attribute returns a matrix whose columns are eigenvectors corresponding to the eigenvalues returned by either valuesV or valuesM attributes.

The eigenvectors and eigenvalues are calculated using an iterative process, with the maximum number of iterations set to 200. The convergence attribute returns the number of iteration it took to the Eigen decomposition process to converge. If the number returned by the convergence attribute equals to 200, then it is likely that the decomposition process needs more than 200 iterations to converge. The number of iterations can be increased to 500 for example, using the sample code below:

```
eig_module:=ML.Mat.Eig(A,500);
eig_valuesV:= eig_module.valuesV;
eig_vectors:= eig_module.vectors;
cnt:=(INTEGER) eig_module.convergence;
```

Lanczos Algorithm

Matrix factorization using the Eigen decomposition is very expensive operation because it is based on an iterative algorithm that converges linearly, while performing an expensive full QR matrix decomposition at every iteration step. A number of methods have been developed to improve convergence rate of the Eigen decomposition by transforming the original matrix into another format which preserves the eigenvalues of the original matrix, and makes the Eigen decomposition converge faster. The most efficient method of that kind is the Lanczos method. It is a technique for converting the original matrix into a tri-diagonal matrix having the same eigenvalues as the original matrix.

The Lanczos method is also popular because it calculates an eigenvalue in every iteration cycle. This feature makes the Lanczos algorithm/method useful in situations where only a few matrix largest or smallest eigenvalues are needed.

The following code demonstrates how to calculate eigenvalues and eigenvectors of matrix A using the Lanczos function to convert a matrix A into its tri-dimensional equivalent before applying the Eigen decomposition to it.

```
IMPORT * FROM ML;
A := dataset([{1,1,12.0},{2,1,6.0},{3,1,4.0},
              {1,2,6.0},{2,2,167.0},{3,2,24.0},
              {1,3,4.0},{2,3,24.0},{3,3,-41.0}], ML.MAT.Types.Element);

T:=ML.Mat.Lanczos(A,3).TComp;
V:=ML.Mat.Lanczos(A,3).VComp;

eigT_val:=ML.Mat.Eig(T).valuesM;
eigT_vec:=ML.Mat.Eig(T).vectors;
eigA_vec := ML.Mat.Mul(V,eigT_vec);
```

The Lanczos function takes 2 arguments, the input matrix to process and the number of eigenvalues to calculate. It produces two results: the TComp attribute returns the tri-diagonal matrix having the same eigenvalues as the original matrix A, and the VComp returns transformation matrix which is needed for calculation of eigenvectors of the original matrix A.

Once the tri-diagonal matrix T gets created, as a result of :

```
T:=ML.Mat.Lanczos(A,3).TComp;
```

The eigenvalues of A can be calculated using the following code:

```
eigA_val:=ML.Mat.Eig(T).valuesM;
```

Notice that since the tri-diagonal matrix T has the same eigenvalues as the original matrix A, eigT_vale=eigA_val.

The eigenvectors of matrix A are calculated by multiplying the V matrix created by the Lanczos algorithm with the eigenvectors of the matrix T.

Singular Value Decomposition (SVD)

Singular value decomposition (SVD) is a factorization of a matrix into a product of three simpler matrices. Formally, the singular value decomposition of a matrix A is a factorization of the form A=USV' where U is an orthogonal matrix, i.e.U is a square matrix with real values whose columns and rows are orthogonal unit vectors, S is a rectangular diagonal matrix with nonnegative real numbers on the diagonal, and V' is a transpose of an orthogonal matrix V.

The orthogonal matrices U and V by definition satisfy the following: U'U = UU' = I, and V'V = VV' = I; The columns of U are orthogonal and unit (i.e. orthonormal) eigenvectors of AA', and the columns of V are orthonormal eigenvectors of A'A. The matrix S is a diagonal matrix containing the square roots of eigenvalues from U or V in descending order.

The diagonal elements of S are known as the singular values of A, and the columns of U and V are called the left singular vectors and right singular vectors of A respectively.

Singular value decomposition components of a matrix U, S and V can be multiplied together to recreate the original matrix exactly. However, if only a subset of rows and columns of matrices U, S, and V are used, then those lower-order matrices U, S, and V provide the best approximation of the original matrix in the least square error sense. Because of that, SVD can be seen as a method for transforming correlated variables represented by columns of the original matrix into a set of uncorrelated variables that better expose relationships that exist among the original data items. SVD can also be used as a method for identifying and ordering the dimensions along which data points exhibit the most variation.

SVD has many applications. For example, SVD could be applied to natural language processing for latent semantic analysis (LSA). LSA starts with a matrix whose rows represent words, columns represent documents, and matrix values (elements) are counts of the word in the document. It then applies SVD to the input matrix, and uses a subset of most significant singular vectors and corresponding singular values to map words and documents into a new space, called 'latent semantic space', where documents are placed near each other measured by co-occurrence of words, even if those words never co-occurred in the training corpus.

The LSA's notion of term-document similarity can be applied to information retrieval, creating a system known as Latent Semantic Indexing (LSI). An LSI system calculates similarity several terms provided in a query have with documents by creating k-dimensional query vector as a sum of k-dimensional vector representations of individual terms, and comparing it to the k-dimensional document vectors.

The code below demonstrates how to use SVD to decompose given matrix A into its U, S and V components:

```
IMPORT * FROM ML;
A := dataset([{1,1,2.0},{1,2,3.0},{1,3,4.0},
              {2,1,2.0}, {2,2,3.0},{2,3,4.0},
              {3,1,2.0},{3,2,3.0}, {3,3,4.0}], ML.Mat.Types.Element);

U := ML.Mat.Svd(A).UComp;
S := ML.Mat.Svd(A).SComp;
V := ML.Mat.Svd(A).VComp;
```

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called 'principal components'. The number of principal components is less than or equal to the number of original variables. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, it accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible, under the constraint that it is orthogonal (meaning uncorrelated with) to the preceding components.

PCA can be done by eigenvalue decomposition of a data covariance matrix or by singular value decomposition of a data matrix, after the matrix vectors (i.e. data features) have been mean adjusted and normalized (Wikipedia).

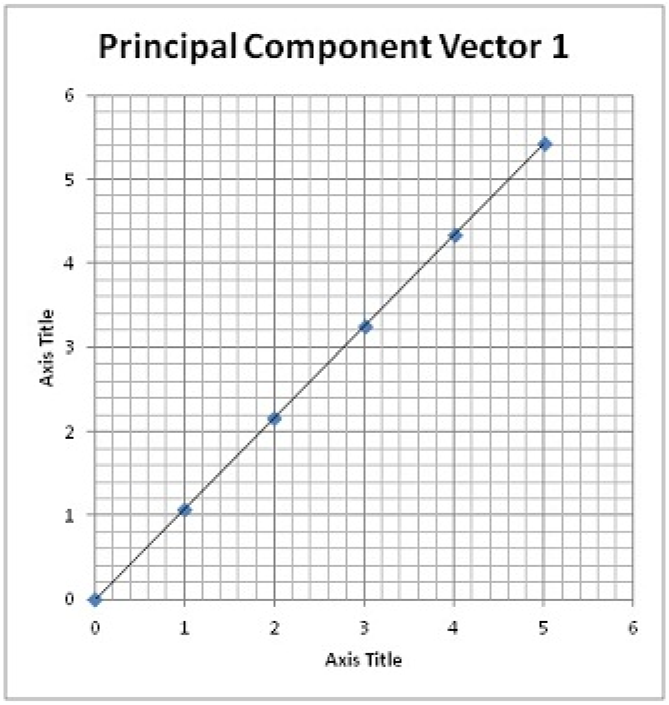
Our PCA implementation is based on a SVD of a data covariance matrix, and it exports two attributes, Ureduce and ZComp. The Ureduce attribute returns a matrix consisting of an ordered set of principal component vectors representing the original data set. In other words, Ureduce is a matrix whose columns are orthogonal vectors (i.e. vectors independent from each other) ordered by their significance in representing features/patterns in the original data set. The ZComp attribute returns matrix which represents projection of the original data set into the new space defined by the Ureduce.

The PCA module takes either one or two arguments. When only one argument is used, it represents the original data set in matrix format, where matrix columns represent individual data features. In that case, the Ureduce attribute returns a matrix containing all principal component vectors associated with the input data set. It is also possible to ignore the components of lesser significance by providing the second argument to the PCA module which represents the number of principal components to preserve.

The following code demonstrates how to use PCA:

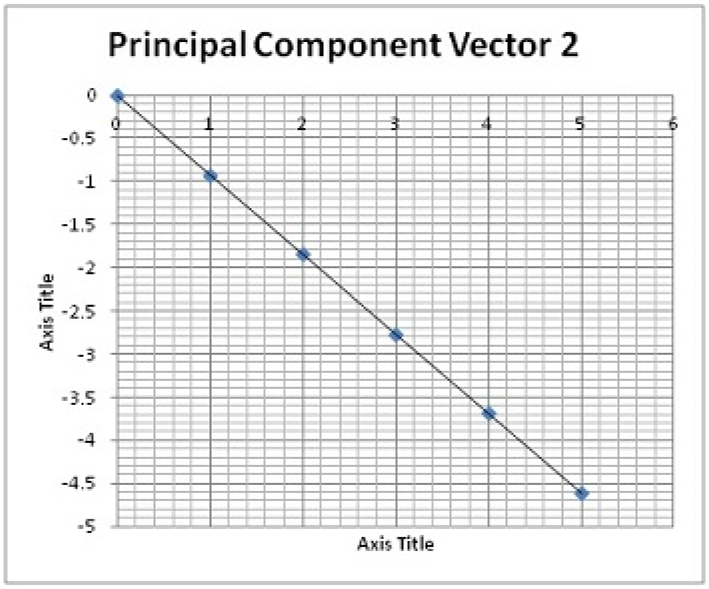
```
IMPORT * FROM ML;
Rec := RECORD
       UNSIGNED rid;
       REAL X1;  // x coordinates
       REAL X2;  // y coordinates
 END;
points := DATASET([{1,2.5,2.4},{2,0.5,0.7},{3,2.2,2.9},{4,1.9,2.2},{5,3.1,3.0},
                   {6,2.3,2.7},{7,2.0,1.6},{8,1.0,1.1},{9,1.5,1.6},{10,1.1,0.9}],Rec);
// Turn into regular NumericField file (with continuous variables)
ToField(points,O);
X := Types.ToMatrix(O(Number in [1,2]));
Ur := Mat.Pca(X).Ureduce;
z := Mat.PCA(X).ZComp;
```

The dataset points represent 10 points as a two-dimensional space (see the Points dataset chart below). This dataset is first converted into its matrix representation X, and then the principal component vectors Ur, and the matrix z as a projection of the matrix X into the Ur space are calculated.



Notice how this vector follows directions of the data pattern, and if we overlaid it onto data points it would go through the middle of data points as if it was the best fit line.

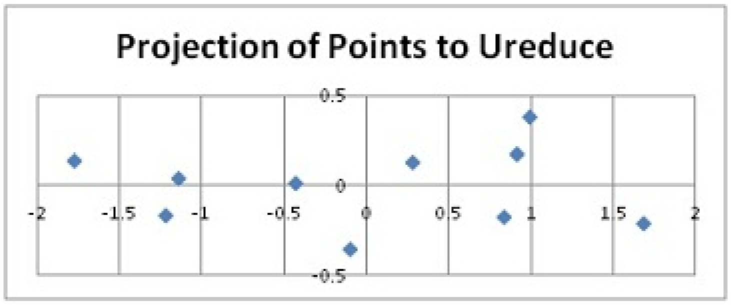
The plot of the second principal component vector looks like this:



This second vector is orthogonal to the first one, and it identifies the following, less important, pattern in the data: while all the points follow the main line, they are off to the sides of the main line by some amount. If we project the original data to the Ur principal component vectors, we get the following results:

X	Y
0.82797	-1.22382
-1.77758	0.142857
0.992197	0.384375
0.27421	0.130417
1.675801	-0.2095
0.912949	0.175282
-0.09911	-0.34982
-1.14457	0.046417
-0.43805	0.017765
-1.22382	-0.16268

These results are shown on data plot chart below. The result of this data projection is equivalent to rotating the principal component vector 1 to make it become horizontal x-axis.



In the example above we decided to use both principal component vectors to transform the original data into the new space, and as a result, we ended up with points in the two-dimensional space with most of data variability along the horizontal x-axis We could also choose to use only the dominant component vector.

We could achieve that by modifying the original code as follows:

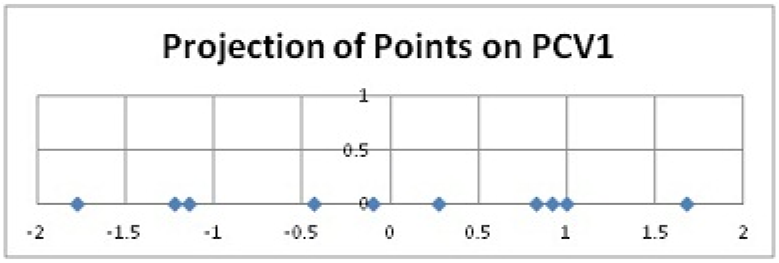
```
Ur := Mat.Pca(X, 1).Ureduce; z := Mat.PCA(X,1).ZComp;
```

Where the second argument '1' indicates that only one (the first) principal component vector is to be used for mapping of the original data set into the new space.

If we project the original data to the first principal component vector, we get the following data:

X
0.82797
-1.77758
0.992197
0.27421
1.675801
0.912949
-0.09911
-1.14457
-0.43805
-1.22382

And the following data plot is produced:



This transformation preserves only the dominant data features, and we can see that the second dimension of the data was removed.

Regression Walk-through

Modules: Regression

The Regression module exists to perform analysis and modeling of the relationship between a single dependent variable Y, and one or more independent variables Xi (also called predictor or explanatory variables). Regression is called ‘Simple’ if only one independent variable X is used. It is called ‘Multivariate’ regression when more than one independent variable is used.

The relationship between dependent variable and independent variables is expressed as a function whose form has to be specified. Regression is called ‘Linear Regression’ if the function that defines the relationship is linear. For example, a Simple Linear Regression model expresses relationship between dependent variable Y and single independent variable X as a linear function:

$Y = \beta 0 + \beta 1X$

This function represents a line with parameters $\beta = (\beta 0, \beta 1)$.

Polynomial

The Polynomial regression expresses (ie models) relationship between dependent variable Y and a single independent variable X with polynomial function of the following form:

$Y = \beta 0 + \beta 1*LogX + \beta 2*X + \beta 3*X*LogX + \beta 4*X2 + \beta 5*X2*LogX + \beta 6*X3$

with parameters:

$\beta = (\beta 0, \beta 1, \beta 2, \beta 3, \beta 4, \beta 5, \beta 6)$

Along with the other ML functions, Polynomial Regression is designed to work upon huge datasets; however it can be quite useful even on tiny ones. The following dataset captures the time taken for a particular ML routine to execute against a particular number of records. Polynomial Regression models relationship between the number of records (X=Recs), and the execution time (Y=Time) and produces β values, which represent how much every component of the polynomial function, (constant, Log, XLogX, …) contributes to the execution time.

```
IMPORT ML;
R := RECORD
       INTEGER        rid;
       INTEGER        Recs;
       REAL           Time;
END;
d := DATASET([{1,50000,1.00},{2,500000,2.29}, {3,5000000,16.15},
               {4,25000000,80.2},{5,50000000,163},{6,100000000,316},
               {7,10,0.83},{8,1500000,5.63}],R);
ML.ToField(d,flds);
P := ML.Regress_Poly_X(flds(number=1),flds(number=2));
P.Beta;
P.RSquared
```

The Polynomial regression is configured by default to calculate parameters $\beta = (\beta 0, \beta 1, \beta 2, \beta 3, \beta 4, \beta 5, \beta 6)$. This can be changed to any number smaller than 6.

For example, to configure polynomial regression to use the polynomial function $Y = \beta 0 + \beta 1*LogX + \beta 2*X$ to model relationship between X and Y, the code above would need to be changed as follows:

```
P := ML.Regress_Poly_X(flds(number=1),flds(number=2), 2);
```

The third parameter ‘2’ has to be used to override the default value ‘6’.

OLS

The Ordinary Least Squares (OLS) regression is a linear regression model that calculates parameters β using the method of least squares to minimize the distance between measured and predicted values of the dependent variable Y. The following example, demonstrates how it might be used:

```
IMPORT ML;

value_record := RECORD
                unsigned    rid;
                unsigned    age;
                real        height;
  END;

d := DATASET([{1,18,76.1}, {2,19,77}, {3,20,78.1},
            {4,21,78.2}, {5,22,78.8}, {6,23,79.7},
            {7,24,79.9}, {8,25,81.1}, {9,26,81.2},
            {10,27,81.8},{11,28,82.8}, {12,29,83.5}]
            ,value_record);

ML.ToField(d,o);

X := O(Number =1); // Pull out the age
Y := O(Number =2); // Pull out the height
Reg := ML.Regression.Sparse.OLS_LU(X,Y);
B := Reg.Beta();
B;
Reg.RSquared;
Reg.Anova;
```

In this example we have a dataset that contains 12 records with the age in months and a mean height in centimeters for children of that age. We use the OLS regression to find parameters β of the linear function, ie the line that represents (models) the relationship between child age and height. Once we get parameters β, we can use them to predict the height of a child whose age is not listed in the dataset.

For example, the mean height of a 30 month old child can be predicted using the following formula:

Height = β0 + 30* β1.

How well does this function (ie regression model) fit the data? One measure of goodness of fit is the R-squared. The range of R-squared is [0,1], and values closer to 1 indicate better fit. For Simple Linear regression the R-squared represents a square of correlation between X and Y.

Analysis of Variance (ANOVA) provides information about the level of variability within a regression model, and that information can be used as a basis for tests of significance. Anova returns a table with the following values for the model, error and total, and the model statistic called “F”:

- sum of squares (SS)
- degrees of freedom (DF)
- mean square (MS)

The OLS regression can be configured to calculate parameters β using either the LU matrix decomposition or Cholesky matrix decomposition.

Visualization Walk-through

The following is a walk-through of the code located in the file VL.Samples. The data used in the code is drawn from the United Nations Statistics Division:

```
lRegions:=RECORD
  STRING continent;
  STRING region;
  UNSIGNED surface_area;
  DECIMAL8_1 pop1950;
  DECIMAL8_1 pop1960;
  DECIMAL8_1 pop1970;
  DECIMAL8_1 pop1980;
  DECIMAL8_1 pop1990;
  DECIMAL8_1 pop2000;
  DECIMAL8_1 pop2010;
END;

dRegions:=DATASET([
  {'Africa','Eastern Africa',6361,64.8,81.9,107.6,143.6,192.8,251.6,324},
  {'Africa','Middle Africa',6613,26.1,32,40.7,53.4,71.7,96.2,126.7},
  {'Africa','Northern Africa',8525,53,67.5,86.9,113.1,146.2,176.2,209.5},
  {'Africa','Southern Africa',2675,15.6,19.7,25.5,33,42.1,51.4,57.8},
  {'Africa','Western Africa',6138,70.5,85.6,107.4,139.8,182.5,235.7,304.3},
  {'Latin America','Caribbean',234,17.1,20.7,25.3,29.7,34.2,38.4,41.6},
  {'Latin America','Central America',2480,37.9,51.7,69.6,91.8,113.2,135.6,155.9},
  {'Latin America','South America',17832,112.4,147.7,191.5,240.9,295.6,347.4,392.6},
  {'North America','North America',21776,171.6,204.3,231.3,254.5,281.2,313.3,344.5},
  {'Asia','Eastern Asia',11763,672.4,801.5,984.1,1178.6,1359.1,1495.3,1574},
  {'Asia','South Central Asia',10791,507.1,620,778.8,986,1246.4,1515.6,1764.9},
  {'Asia','South Eastern Asia',4495,172.9,219.3,285.2,359,445.4,523.8,593.4},
  {'Asia','Western Asia',4831,51,66.8,86.9,114,148.6,184.4,232},
  {'Europe','Eastern Europe',18814,220.1,252.8,276.2,294.9,310.5,304.2,294.8},
  {'Europe','Northern Europe',1810,78,81.9,87.4,89.9,92.1,94.3,99.2},
  {'Europe','Southern Europe',1317,108.3,117.4,126.8,137.7,142.4,145.1,115.2},
  {'Europe','Western Europe',1108,140.8,151.8,165.5,170.4,175.4,183.1,189.1},
  {'Oceania','Australia and New Zealand',8012,10.1,12.7,15.5,17.9,20.5,23,26.6},
  {'Oceania','Melanesia',541,2.2,2.6,3.3,4.3,5.5,7,8.7},
  {'Oceania','Micronesia',3,.1,.2,.2,.3,.4,.5,.5},
  {'Oceania','Polynesia',8,.2,.3,.4,.5,.5,.6,.7}
],lRegions);
```

In our first set of charts, we are going to slim this dataset down to the continent level and report on the surface area for each. This is accomplished with a simple TABLE command. Once we have the data we are looking for, we can translate it into the VL.Types.ChartData format using the FormatData macro:

```
dContinentArea:=TABLE(dRegions,{
  continent;
  UNSIGNED surface_area:=SUM(GROUP,surface_area);
},continent);
dContinentChart:=VL.FormatData(dContinentArea,continent);
```

We will use all the default styles, except that we will add a meaningful title:

```
ContinentStyle:=VL.Styles.SetValue(Vl.Styles.Default,title,'Surface Area by Continent');
```

Once we have these components, we have all we need to produce some very quick and easy charts, such as the Pie, Bar and Column shown below:

```
VL.Chart('SurfaceAreaPie',dContinentChart,ContinentStyle).Pie;
VL.Chart('SurfaceAreaBar',dContinentChart,ContinentStyle).Bar;
VL.Chart('SurfaceAreaColumn',dContinentChart,ContinentStyle).Column;
```

In a slightly more advanced example, we will now present time-oriented population statistics. As with the above example, the first step is to re-shape the data into just the fields we want, by removing the extraneous continent and surface_area fields. Then we can transform the result into our ChartData format.

```
dPopOnly:=PROJECT(dRegions,{lRegions AND NOT [continent,surface_area]});
dPopData:=VL.FormatData(dPopOnly,region);
```

At this point, if we were to paint the charts we would have the data points on the wrong axis and the visualization would be relatively meaningless for our purposes. So we run the formatted dataset through the function "SwapAxes", which (as the name implies) swaps the X and Y axes. Since the X-Axis was originally a series of columns, the user needs to name the new X-Axis, which is the purpose of the second parameter:

```
dSwapped:=VL.SwapAxes(dPopData,'Decade');
```

Now we may continue producing meaningful charts, such as the line and area charts below. A call to the SetValue Styles function like in the above example adds in a title as before:

```
PopStyle:=VL.Styles.SetValue(Vl.Styles.Default,title,'Population by Region over Time');
VL.Chart('PopulationByRegionLine',dSwapped,PopStyle).Line;
VL.Chart('PopulationByRegionArea',dSwapped,PopStyle).Area;
```

One slightly more advanced chart the one can use from Google's arsenal is the Combo chart. This one enables the user to specify two different types of charts that overlay each other. In the below example, we will plot a bar graph of the 2010 population for the four regions of Africa, and include a line graph showing the overall average of that population across all countries. In this case, we are going to filter our dataset on Africa, and then massage our data in the same way as above to remove unwanted fields.

In addition to this subset, we also need to add in a row containing the average, which we do in the row directly following our filter.

```
dAfrica:=PROJECT(dRegions(continent='Africa'),{lRegions AND NOT [continent,surface_area]});
dWithAverages:=dAfrica+TABLE(dAfrica,{
  STRING region:='Average';
  DECIMAL8_1 pop1950:=AVE(GROUP,pop1950);
  DECIMAL8_1 pop1960:=AVE(GROUP,pop1960);
  DECIMAL8_1 pop1970:=AVE(GROUP,pop1970);
  DECIMAL8_1 pop1980:=AVE(GROUP,pop1980);
  DECIMAL8_1 pop1990:=AVE(GROUP,pop1990);
  DECIMAL8_1 pop2000:=AVE(GROUP,pop2000);
  DECIMAL8_1 pop2010:=AVE(GROUP,pop2010);
});
```

Once we have built our 5-row table, we can now perform the SwapAxes transform. However, we still have one more thing to do that is a little more difficult. Producing a combo Chart requires that parameters be sent to Google that are not standard enough to be part of our default set of style parameters. As such, we need to make use of the ChartAdvanced attribute of the Styles module. This field exists so that the user may pass through specific exact strings as parameters.

Such a task requires that the user have specific knowledge of the API they are calling, but with that knowledge the user has complete control over all of the buttons and knobs that are available in any of the API's. In the SetValue call below, we are setting the title to something meaningful, we are defaulting the chart to a Bar type, and then changing the fifth series from Bar to Line. If the user desires more detail on this type of control, it is a good idea to go to the web site of the API and read through the documentation regarding what options are available.

```
dFormatted:=VL.SwapAxes(VL.FormatData(dWithAverages,region),'Decade');
ComboStyle:=Vl.Styles.SetValue(Vl.Styles.Default,ChartAdvanced,'title:"Population Growth in
Africa",seriesType:"bars",series:{5:{type:"line"}}');
VL.Chart('AfricanPopulationGrowthCombo',dFormatted,ComboStyle).Combo;
```

The ML Modules

Each ML module focuses on a specific type of algorithm and contains a number of routines. The functionality of each routine is also described.

Performance statistics are provided for some routines. These were carried out on a 10 node cluster and are for comparison purposes only.

Associations (ML.Associate)

Use this module to perform frequent pattern matching on the underlying data. The following routines are provided:

Routine	Description
Apriori1,Apriori2,Apriori3	Uses 'old school' brute force and speed approach to produce patterns of up to 3 items which appear together with a particular degree of support.
AprioriN	Uses 'new school' techniques finding all patterns of up to N items, appearing together with a particular degree of support.
EclatN	Uses the 'eclat' technique to construct a result which is identical to AprioriN.
Rules	Uses patterns generated by AprioriN or EclatN to answer the question: "given a group of M items exists; what is the M+1th most likely to be".

The following performance statistics of these routines were observed using a 10 node cluster:

Routine	Description	Result	Expected Order
Apriori1	On 140M words	47 secs	NlgN
Aprior1	On 197M words	91 secs	NlgN
Apiori2	On 140M words, producing 2.6K pairs	325 secs	(N/k)^2.MLg(N) where k is the proportion of 'buckets' the average item is in. (Using terms in 5-10% of buckets)
Apiori2	On 193M words (using .1->1% buckets) producing 4.4M pairs	21 mins	
Apiori2	On 19M words (10% sample), producing 4.1M pairs	2 mins	
Apiori3	On 140M words (terms in 5-10% buckets)	Exploded	
Apiori3	1.9M words (1% sample of .1-1 buckets), (172K possible 3 groups), 3.6B intermediate results, 22337 eventual results	73 mins	
Apiori3	On 1.9M words with new LOOKUP optimization	42 mins	
EE3	On 1.9M words (1% sample of .1-1 buckets), 22337 eventual results	3 mins	
EE10	On 1.9M words	Locks	

This example shows how to use the Apriori routine in conjunction with the Doc module. A simpler example can be found in the Association walk-through.

```
IMPORT ML;
IMPORT ML.Docs AS Docs;

d11 := DATASET([{'One of the wonderful things about tiggers is tiggers are wonderful
                 things'},
                {'It is a little scary the drivel that enters ones mind when given the
                 task of entering random text'},
                {'I almost quoted Oscar Wilde; but I considered that I had gotten a little
                 too silly already!'},
                {'I would hate to have quoted silly people!'},
                {'Oscar Wilde is often quoted'},
                {'In Hertford, Hereford and Hampshire Hurricanes hardly ever happen'},
                {'It is a far, far better thing that I do, than I have ever done'}],
                {string r});
d00 := DATASET([{'aa bb cc dd ee'},{'bb cc dd ee ff gg hh ii'},{'bb cc dd ee ff gg hh ii'},
                {'dd ee ff'},{'bb dd ee'}],{string r});

d := d11;
d1  := PROJECT(d,TRANSFORM(Docs.Types.Raw,SELF.Txt := LEFT.r));
d2  := Docs.Tokenize.Enumerate(d1);
d3  := Docs.Tokenize.Clean(d2);
d4  := Docs.Tokenize.Split(d3);

lex := Docs.Tokenize.Lexicon(d4);

o1  := Docs.Tokenize.ToO(d4,lex);
o2  := Docs.Trans(O1).WordBag;

lex;
ForAssoc  := PROJECT( o2, TRANSFORM(ML.Types.ItemElement,SELF.id := LEFT.id,
SELF.value  := LEFT.word ));
ForAssoc;
ML.Associate(ForAssoc,2).Apriori1;
ML.Associate(ForAssoc,2).Apriori2;
ML.Associate(ForAssoc,2).Apriori3;
ML.Associate(ForAssoc,2).AprioriN(40);
```

Example - Using ML.Associate on a One-Click^{TM} Thor

The HPCC One-Click^{TM} Thor system is very quick and easy to setup and provides the perfect test environment for starting to use the HPCC ML Libraries. It is assumed that you have already installed the ECL IDE and the ML Module (which includes the example code needed). The data used is based on character information from Marvel comics and is taken from the Amazon Web Service Public Data area. It is available for immediate use by simply adding it to your One-Click^{TM} Thor as a snapshot on launching the cluster.

Setting up the One-Click^{TM} Thor

You must have an Amazon AWS account to be able to setup a One-Click^{TM} Thor cluster. To setup your One-Click^{TM} Thor cluster:

1. Go to the One-Click^{TM} Thor **Getting Started** page on the HPCC Systems website: https://aws.hpccsystems.com/aws/getting_started/ and following the **Logging In** instructions.

2. Click **Launch Cluster**.

3. Select the **Region**, which for the example data is **Virginia**.

4. Enter the number of **Thor Nodes** you want (1 node is sufficient for this example).

5. Enter the **Snapshot ID** for the example data as **snap-7766d116**.

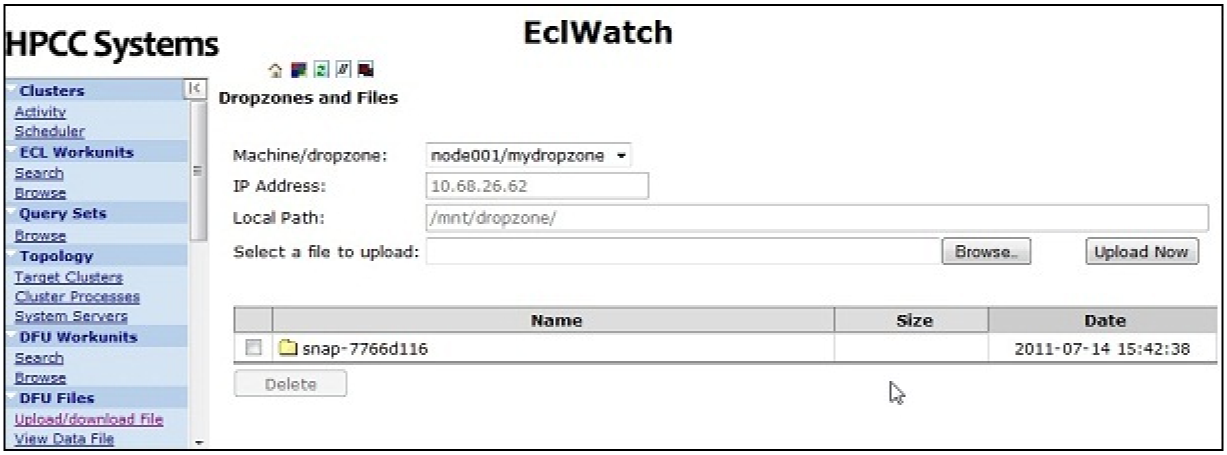
6. Press **Launch Cluster**.

Note: To add a snapshot of data from the Amazon Public Data Set , you need to know which **Region** it is linked with.

The **Launch Log** shows the status of your One-Click^{TM} Thor. When the configuration process is complete, the IP address of the ESP is shown in a link at the bottom of the page (xx.xx.xx.xxx.8010). Click on this link to access the **ECL Watch** page.

Using **ECL Watch**, click on the **DFU Files/Upload/download File** menu option to view the contents of the **Dropzone**.

The snapshot we attached to the One-Click^{TM} Thor system is shown in the list and is ready to be used. Double-click on the folder to view the files which are included with the snapshot.



Dropzones and Files

Machine/dropzone: node001/mydropzone

IP Address: 10.68.26.62

Local Path: /mnt/dropzone/

Select a file to upload:

	Name	Size	Date
□	snap-7766d116		2011-07-14 15:42:38

Note: You can use any data that is available on Amazon as well as any data that you have made and published as part of the Amazon Public Data Set. (If you want to know more about how to do this see, http://aws.amazon.com/publicdatasets/.) You can also upload your own data from your own machine using the **DFU Files/Upload/download File** menu option in **ECL Watch** shown above.

Running the example using ECL IDE

The ESP IP address changes every time you launch a new One-Click^{TM} Thor cluster which means that you need to add a new ECL IDE configuration. On starting ECL IDE, click the **Preferences** button and enter the public ESP IP address in the **Server** field. Click **OK** to return to the login window, where you can select your new configuration and login.

Note: The public ESP IP address is available on the One-Click^{TM} Thor **Launch Log**.

The example ECL file is called **Association_Marvel_Comics.ecl** and is located by expanding the tree in the **Repository Window** as follows:

- **My Files**/**ML/Tests/Explanatory**

Open this file either by double-clicking on it or by using the **Open in Builder Window** right-click option.

This file uses the Apriori3 and EclatN routines from the Associations Machine Learning Module to create character associations using the snapshot data (Marvel Universe Social Graph). There are some changes that need to be made to this ECL file:

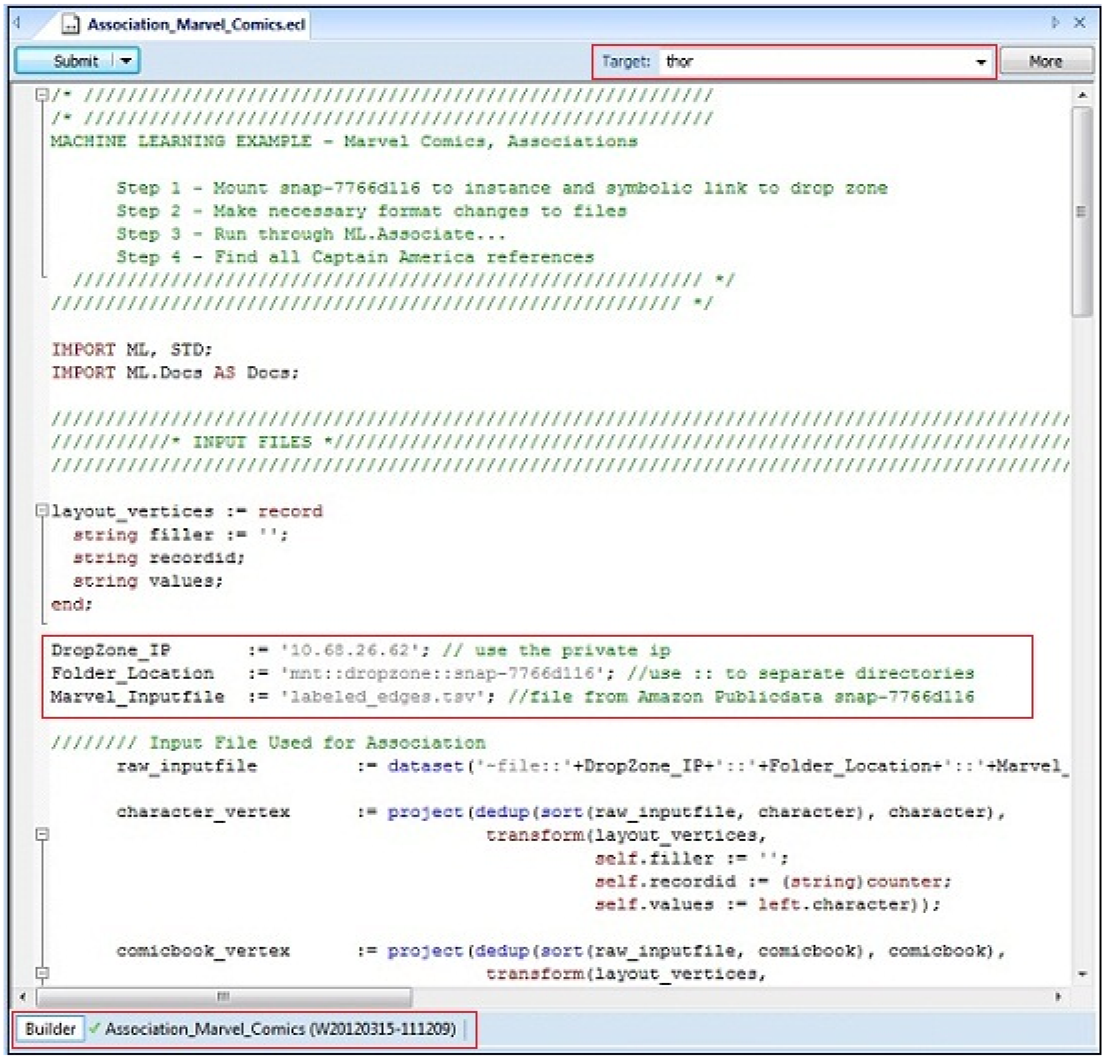
1. Change the **Dropzone_IP** address to use the private IP address. This information is available either from the One-Click^{TM} Thor **Launch Log** or by using the **System Servers** menu item in **ECL Watch**.

2. Change the **Folder_Location** to the location of the snapshot, (e.g. **mnt::dropzone::snap-7766d116).**

3. Change the **Marvel_Inputfile** name to the exact name of the file on the **Dropzone**. (In this case the file is called. **labeled_edges.tsv**)

4. Select **thor** as the **Target**, and **Submit**.

A green tick indicates that the job has completed. View the results by clicking on the completed workunit.

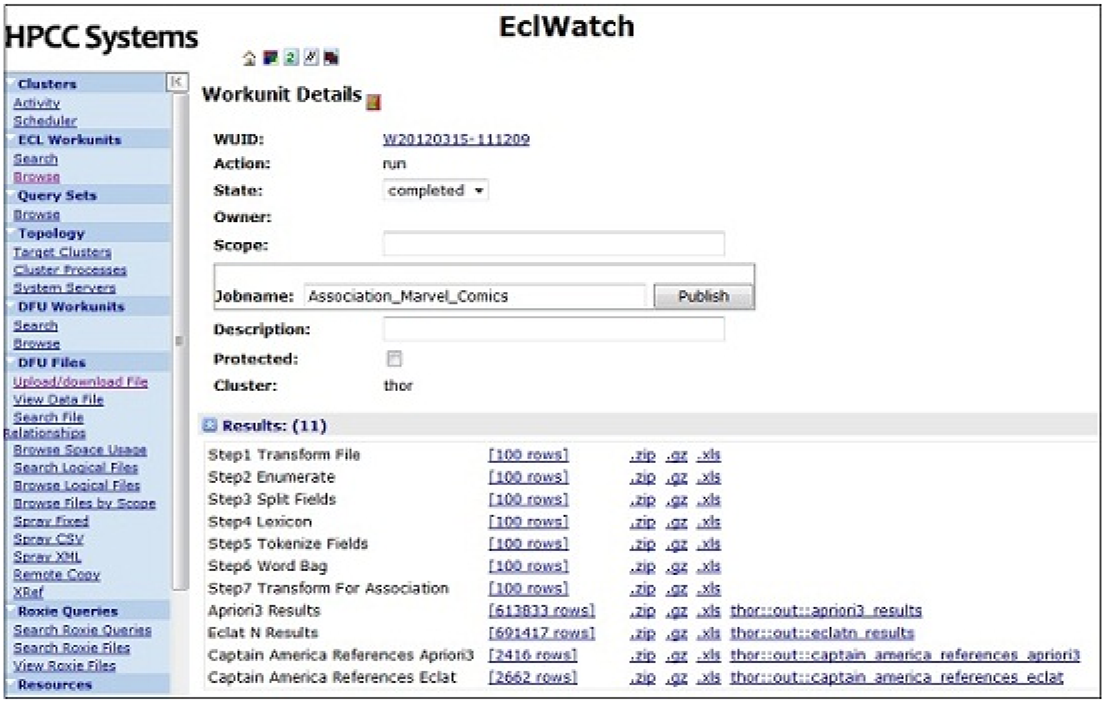

Association_Marvel_Comics.ecl
Submit
Target: thor
More
Builder
Association_Marvel_Comics (W20120315-111209)

```
/* //////////////////////////////////////////////////////////
/* //////////////////////////////////////////////////////////
MACHINE LEARNING EXAMPLE - Marvel Comics, Associations

      Step 1 - Mount snap-7766d116 to instance and symbolic link to drop zone
      Step 2 - Make necessary format changes to files
      Step 3 - Run through ML.Associate...
      Step 4 - Find all Captain America references
  //////////////////////////////////////////////////////////// */
//////////////////////////////////////////////////////////// */

IMPORT ML, STD;
IMPORT ML.Docs AS Docs;

///////////////////////////////////////////////////////////////////////////////////////////////
///////////* INPUT FILES *///////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////////////////////

layout_vertices := record
  string filler := '';
  string recordid;
  string values;
end;

DropZone_IP      := '10.68.26.62'; // use the private ip
Folder_Location  := 'mnt::dropzone::snap-7766d116'; //use :: to separate directories
Marvel_Inputfile := 'labeled_edges.tsv'; //file from Amazon Publicdata snap-7766d116

//////// Input File Used for Association
      raw_inputfile         := dataset('~file::'+DropZone_IP+'::'+Folder_Location+'::'+Marvel_

      character_vertex      := project(dedup(sort(raw_inputfile, character), character),
                                  transform(layout_vertices,
                                            self.filler := '';
                                            self.recordid := (string)counter;
                                            self.values := left.character));

      comicbook_vertex      := project(dedup(sort(raw_inputfile, comicbook), comicbook),
                                  transform(layout_vertices,
```

Interpreting the results

Using ECL Watch you can also view the progress, details and results of a workunit. It is a matter of preference whether you choose to use ECL IDE or ECL Watch. For the purposes of this example, results are shown using ECL Watch:

1. Click on the **ECL Workunits/Browse** menu item and locate your workunit in the list.
2. Click on the **WUID** link to display the **Workunit Details** page.
3. Expand the **Results** link. Click on the rows link for any result to see the values



Workunit Details

WUID: W20120315-111209
Action: run
State: completed
Owner:
Scope:
Jobname: Association_Marvel_Comics
Description:
Protected: □
Cluster: thor

Results: (11)

Step1 Transform File	[100 rows]	.zip .gz .xls	
Step2 Enumerate	[100 rows]	.zip .gz .xls	
Step3 Split Fields	[100 rows]	.zip .gz .xls	
Step4 Lexicon	[100 rows]	.zip .gz .xls	
Step5 Tokenize Fields	[100 rows]	.zip .gz .xls	
Step6 Word Bag	[100 rows]	.zip .gz .xls	
Step7 Transform For Association	[100 rows]	.zip .gz .xls	
Apriori3 Results	[613833 rows]	.zip .gz .xls	thor::out::apriori3_results
Eclat N Results	[691417 rows]	.zip .gz .xls	thor::out::eclatn_results
Captain America References Apriori3	[2416 rows]	.zip .gz .xls	thor::out::captain_america_references_apriori3
Captain America References Eclat	[2662 rows]	.zip .gz .xls	thor::out::captain_america_references_eclat

The results produced reflect the Apriori3 and Eclat(N) example build process and are interpreted as follows:

Result	Description
Transform File	Initial layout adding ID field to enumerate rows.
Enumerate	Rows are numbered in sequential order.
Split Fields	Normalize and number the position of "word" in each row.
Lexicon	Uses the Split Fields results to generate the total for each "word" and in how many "docs".
Tokenize Fields	Adds new Lexicon fields to the Split Fields result.
WordBag	Finds the maximum amount of words and docs and counts the words in each doc.
Transform for Association	Transform into layout that is usable for creating the associations.
Apriori3	ML.Associate([input dataset],[minimum supports]).Apriori3; Uses ‘old school’ brute force and speed approach to produce patterns of up to 3 items which appear together with a particular degree of support.
EclatN(3)	ML.Associate([input dataset],[minimum supports]).EclatN([max item patterns]) Uses the ‘eclat’ technique for finding all patterns of up to N items, appearing together with a particular degree of support. Gives the exact same results as AprioriN.

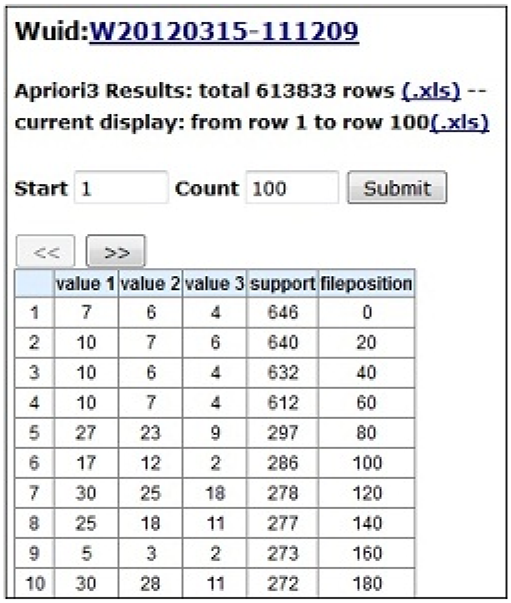
Apriori3 and EclatN(3) example results

The Marvel Comics Social Graph data uses vertices that have numeric codes to represent names and comic books. The Find Captain America results contains both the code and code translations.

The results shown include the top 10 results only.

Apriori 3 Association Results

On the results page, click on the rows link for the **Apriori 3 Results** to view the following:



Wuid:W20120315-111209

Apriori3 Results: total 613833 rows (.xls) -- current display: from row 1 to row 100(.xls)

Start 1 **Count** 100 Submit

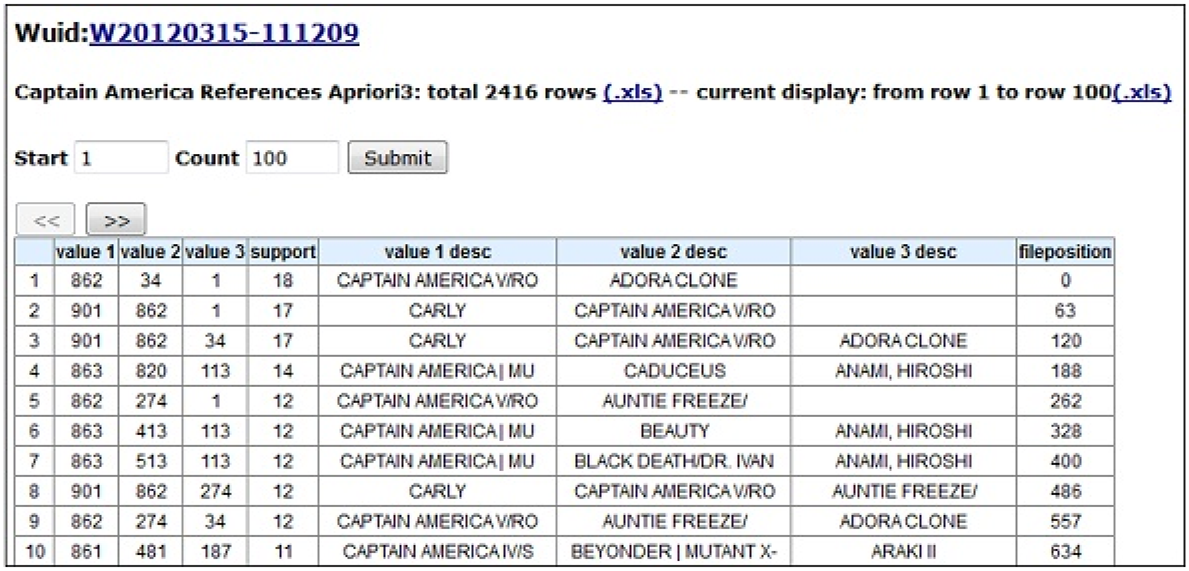
	value 1	value 2	value 3	support	fileposition
1	7	6	4	646	0
2	10	7	6	640	20
3	10	6	4	632	40
4	10	7	4	612	60
5	27	23	9	297	80
6	17	12	2	286	100
7	30	25	18	278	120
8	25	18	11	277	140
9	5	3	2	273	160
10	30	28	11	272	180

Values 1, **2** and **3** contain the codes given in the input file.

Support shows the number of occurances where the association relationship was found.

Apriori3 Find Captain America Results

On the results page, click on the rows link for the **Captain America References Apriori3** to view the following:



Wuid:W20120315-111209

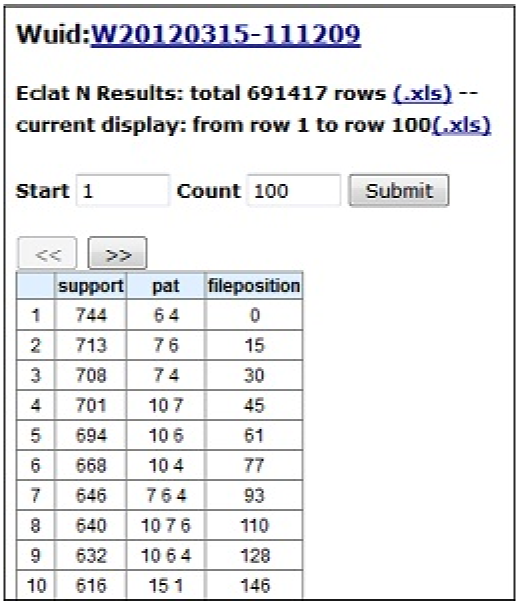
Captain America References Apriori3: total 2416 rows (.xls) -- current display: from row 1 to row 100(.xls)

Start 1 **Count** 100 Submit

	value 1	value 2	value 3	support	value 1 desc	value 2 desc	value 3 desc	fileposition
1	862	34	1	18	CAPTAIN AMERICA V/RO	ADORA CLONE		0
2	901	862	1	17	CARLY	CAPTAIN AMERICA V/RO		63
3	901	862	34	17	CARLY	CAPTAIN AMERICA V/RO	ADORA CLONE	120
4	863	820	113	14	CAPTAIN AMERICA | MU	CADUCEUS	ANAMI, HIROSHI	188
5	862	274	1	12	CAPTAIN AMERICA V/RO	AUNTIE FREEZE/		262
6	863	413	113	12	CAPTAIN AMERICA | MU	BEAUTY	ANAMI, HIROSHI	328
7	863	513	113	12	CAPTAIN AMERICA | MU	BLACK DEATH/DR. IVAN	ANAMI, HIROSHI	400
8	901	862	274	12	CARLY	CAPTAIN AMERICA V/RO	AUNTIE FREEZE/	486
9	862	274	34	12	CAPTAIN AMERICA V/RO	AUNTIE FREEZE/	ADORA CLONE	557
10	861	481	187	11	CAPTAIN AMERICA IV/S	BEYONDER | MUTANT X-	ARAKI II	634

EclatN(3) Association Results

On the results page, click on the rows link for the **Eclat N Results** to view the following:



Wuid:W20120315-111209

Eclat N Results: total 691417 rows (.xls) -- current display: from row 1 to row 100(.xls)

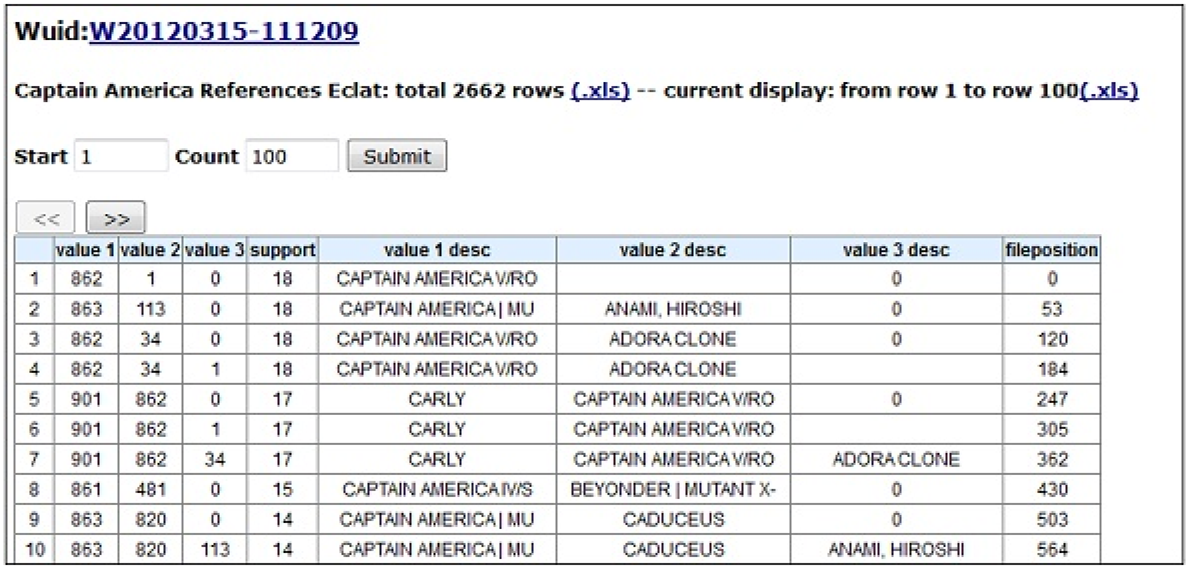
	support	pat	fileposition
1	744	6 4	0
2	713	7 6	15
3	708	7 4	30
4	701	10 7	45
5	694	10 6	61
6	668	10 4	77
7	646	7 6 4	93
8	640	10 7 6	110
9	632	10 6 4	128
10	616	15 1	146

Pat contains the codes given in the input file.

Support shows the number of occurances where the association relationship was found.

EclatN(3) Find Captain America Results

On the results page, click on the rows link for the **Captain America References Eclat** results to view the following:



Wuid:W20120315-111209

Captain America References Eclat: total 2662 rows (.xls) -- current display: from row 1 to row 100(.xls)

	value 1	value 2	value 3	support	value 1 desc	value 2 desc	value 3 desc	fileposition
1	862	1	0	18	CAPTAIN AMERICA V/RO		0	0
2	863	113	0	18	CAPTAIN AMERICA | MU	ANAMI, HIROSHI	0	53
3	862	34	0	18	CAPTAIN AMERICA V/RO	ADORA CLONE	0	120
4	862	34	1	18	CAPTAIN AMERICA V/RO	ADORA CLONE		184
5	901	862	0	17	CARLY	CAPTAIN AMERICA V/RO	0	247
6	901	862	1	17	CARLY	CAPTAIN AMERICA V/RO		305
7	901	862	34	17	CARLY	CAPTAIN AMERICA V/RO	ADORA CLONE	362
8	861	481	0	15	CAPTAIN AMERICA IV/S	BEYONDER | MUTANT X-	0	430
9	863	820	0	14	CAPTAIN AMERICA | MU	CADUCEUS	0	503
10	863	820	113	14	CAPTAIN AMERICA | MU	CADUCEUS	ANAMI, HIROSHI	564

Classify (ML.Classify)

Use this module to tackle the problem, “can I predict this dependent variable based upon these independent ones?”. The following routines are provided:

Routine	Description
BuildNaiveBayes	Builds a Bayes model for one or more dependent variables.
NaiveBayes	Executes one or more Bayes models against an underlying dataset to compute dependent variables.
TestNaiveBayes	Generates a module containing four different measures of how well the classification models are doing. This calculation is based on the Bayes model and set of independent variables with outcome data supplied.
BuildPerceptron	Builds a perceptron for multiple dependent (Boolean) variables.

Cluster (ML.Cluster)

This module is used to perform the clustering of a collection of records containing fields.

The following routines are provided:

Routine	**Description**
DF	A submodule which is used to perform various distance metrics upon two records. Currently, the following are provided: • Euclidean • Euclidean Squared • Manhattan • Cosine • Tanimoto Quick variants exist for both Euclidean and Euclidean Squared which are much faster on sparse data PROVIDED you are willing to accept no distance, if there are no dimensions along which the vectors touch.
Distances	The engine to actually compute the distance matrix (as a matrix).
Closest	Takes a set of distances and returns the closest centroid for each row.
KMeans	Performs KMeans clustering for a specified number of iterations.
AggloN	Performs Agglomerative (Hierarchical) clustering. The results include cluster assignments, remaining distances between clusters and even the dentrogram.

KMeans

The KMeans module performs K-Means clustering on a static primary data set and a pre-determined set of centroids.

This is an iterative two-step process. For each iteration, every data entity is assigned to the centroid that is closest to it, and then the centroid locations are re-assigned to the mean position of all the data entities assigned to them.

The user passes in the two datasets and the maximum number of iterations to perform if convergence is not achieved.

Optionally, the user may also define a convergence threshold (default=0.0) and the distance function to be used during calculations (default is Euclidean).

The primary output from the KMeans module is “Result()”, which returns the centroid dataset with their new locations. “Convergence” will indicate the number of iterations performed, which will never be greater than the maximum number passed in to the function.

There are also a number other useful return values that enable the user to analyze centroid movement. These are included in the following example.

```
IMPORT ML;

lMatrix:={UNSIGNED id;REAL x;REAL y;};

// Simple two-dimensional set of numbers between 0-10
dDocumentMatrix:=DATASET([

  {1,2.4639,7.8579},
  {2,0.5573,9.4681},
  {3,4.6054,8.4723},
  {4,1.24,7.3835},
  {5,7.8253,4.8205},
  {6,3.0965,3.4085},
  {7,8.8631,1.4446},
  {8,5.8085,9.1887},
  {9,1.3813,0.515},
  {10,2.7123,9.2429},
  {11,6.786,4.9368},
  {12,9.0227,5.8075},
  {13,8.55,0.074},
  {14,1.7074,3.9685},
  {15,5.7943,3.4692},
  {16,8.3931,8.5849},
  {17,4.7333,5.3947},
  {18,1.069,3.2497},
  {19,9.3669,7.7855},
  {20,2.3341,8.5196}
],lMatrix);

// Arbitrary but non-symmetric centroid starting points
dCentroidMatrix:=DATASET([
  {1,1,1},
  {2,2,2},
  {3,3,3},
  {4,4,4}
],lMatrix);

// Convert the above matrices into the NumericField format
ML.ToField(dDocumentMatrix,dDocuments);
ML.ToField(dCentroidMatrix,dCentroids);

// Set up KMeans with a maximum of 30 iterations and .3 as a convergence threshold
KMeans:=ML.Cluster.KMeans(dDocuments,dCentroids,30,.3);

// The table that contains the results of each iteration
KMeans.Allresults;
// The number of iterations it took to converge
KMeans.Convergence;
// The results of iteration 3
KMeans.Result(3);
// The distance every centroid travelled across each axis from iterations 3 to 5
KMeans.Delta(3,5);
// The total distance the centroids travelled on each axis
KMeans.Delta(0);
// The straight-line distance travelled by each centroid from iterations 3 to 5
KMeans.DistanceDelta(3,5);
// The total straight-line distance each centroid travelled
KMeans.DistanceDelta(0);
// The distance travelled by each centroid during the last iteration.
KMeans.DistanceDelta();
```

AggloN

The AggloN function performs Agglomerative clustering on a static document set. This is a bottom-up approach whereby close pairs are found and linked, and then treated as a single entity during the next iteration. Allowed to run fully, the result will be a single tree structure with multiple branches and sub-branches.

The dataset and the maximum number of iterations are the two required parameters for this function. The user may also optionally specify the distance formula which defaults to Euclidean. The following is an example using the AggloN routine (using the same dDocuments constructed in the above example):

```
// Set up Agglomerative clustering with 4 iterations
AggloN:=ML.Cluster.AggloN(dDocuments,4);

// Table with nested sets of numbers delineating the tree after the specified
// number of iterations
AggloN.Dendrogram;
// Table containing the distances between each pair of clusters
AggloN.Distances;
// List of entities and the cluster they are assigned to (the cluster ID will always
// be the lowest ID value of the entities that comprise the cluster)
AggloN.Clusters;
```

Distances

This is the engine for calculating the distances between every entity in one dataset to every entity in a second dataset, which are the two required parameters. If the first dataset is also passed as the second one, then a self-join is performed. Otherwise, it is assumed that the IDs for the second dataset do not intersect with those from the first one. The default formula used for distance calculation is Euclidean, but that may be changed to any of the other distance functions available in the DF module.

The code for the Distances function is fairly complex in an effort to facilitate both flexibility and efficiency. The user is able to calculate distances in three modes:

Routine	**Description**
Dense	All calculations are performed, which for a self-join is an N^2 problem.
Summary Join	Highly efficient but not as accurate, this process only calculates distance between entities that share at least one dimension.
Background	Fully accurate and highly efficient, assuming a sparse matrix. This process takes advantage of the assumption that only a small number of dimensions are shared between any two pairs of entities.

This is fully accurate and highly efficient, assuming a sparse matrix. This process takes advantage of the assumption that only a small number of dimensions are shared between any two pairs of entities.

Correlations (ML.Correlate)

Use this module to calculate the degree of correlation between every pair of fields provided. The following routines are provided:

Routine	**Description**
Simple	Pearson and Spearman correlation co-efficients for every pair of fields.
Kendal	Kendal’s Tau for every pair of fields.

The correlation tables expose multicollinearity issues and enable you to look at predicted versus original to expose heteroscedasticity issues.

Discretize (ML.Discretize)

This module provides a suite of routines which allow a datastream with continuous real elements to be turned into a stream with discrete (integer) elements. The Discretize module currently supports three methods of discretization, ByRounding, ByBucketing and ByTiling. These method can be used by hand for reasons of simplicity and control.

In addition, it is possible to turn them into an instruction stream for an 'engine' to execute. Using them in this way allows the discretization strategy to be in meta-data. Use the Do definition to construct the meta-data fragment, which will then perform the discretization. This enables the automatic generation of strategies and even iterates over the modeling process with different discretization strategies which can be programmatically generated. The following routines are provided:

Routine	Description
ByRounding	Using scale and delta.
ByBucketing	Split the range evenly and distribute the values potentially unevenly.
ByTiling	Split the values evenly and have an uneven range.
Do	Constructs a meta-data fragment which will then perform the discretization.

The following is an example of the use of the Naive Bayes routine:

```
import ml;

value_record := RECORD
        unsigned rid;
        real height;
        real weight;
        real age;
        integer1 species;
        integer1 gender; // 0 = unknown, 1 = male, 2 = female
END;
d := dataset([{1,5*12+7,156*16,43,1,1},
              {2,5*12+7,128*16,31,1,2},
              {3,5*12+9,135*16,15,1,1},
              {4,5*12+7,145*16,14,1,1},
              {5,5*12-2,80*16,9,1,1},
              {6,4*12+8,72*16,8,1,1},
              {7,8,32,2.5,2,2},
              {8,6.5,28,2,2,2},
              {9,6.5,28,2,2,2},
              {10,6.5,21,2,2,1},
              {11,4,15,1,2,0},
              {12,3,10.5,1,2,0},
              {13,2.5,3,0.8,2,0},
              {14,1,1,0.4,2,0}
              ]
              ,value_record);
// Turn into regular NumericField file (with continuous variables)
ml.ToField(d,o);

// Hand-code the discretization of some of the variables
disc := ML.Discretize.ByBucketing(o(Number IN [2,3]),4)+ML.Discretize.ByTiling(o(Number IN
[1]),6)+ML.Discretize.ByRounding(o(Number=4));

// Create instructions to be executed
inst :=
ML.Discretize.i_ByBucketing([2,3],4)+ML.Discretize.i_ByTiling([1],6)+
ML.Discretize.i_ByRounding([4,5]);
```

```
// Execute the instructions
done := ML.Discretize.Do(o,inst);

//m1 := ML.Classify.BuildPerceptron(done(Number<=3),done(Number>=4));
//m1

m1 := ML.Classify.BuildNaiveBayes(done(Number<=3),done(Number>=4));
m1;

Test := ML.Classify.TestNaiveBayes(done(Number<=3),done(Number>=4),m1);
Test.Raw;
Test.CrossAssignments;
Test.PrecisionByClass;
Test.Headline;
```

The following is an example of the use of the discretize routines:

```
import ml;
value_record := RECORD
                unsigned rid;
                real height;
                real weight;
                real age;
                integer1 species;
END;
d := dataset([{1,5*12+7,156*16,43,1},
              {2,5*12+7,128*16,31,1},
              {3,5*12+9,135*16,15,1},
              {4,5*12+7,145*16,14,1},
              {5,5*12-2,80*16,9,1},
              {6,4*12+8,72*16,8,1},
              {7,8,32,2.5,2},
              {8,6.5,28,2,2},
              {9,6.5,28,2,2},
              {10,6.5,21,2,2},
              {11,4,15,1,2},
              {12,3,10.5,1,2},
              {13,2.5,3,0.8,2},
              {14,1,1,0.4,2}
              ]
              value_record);
// Turn into regular NumericField file (with continuous variables)
ml.macPivot(d,o);

// Hand-code the discretization of some of the variables
disc := ML.Discretize.ByBucketing(o(Number = 3),4)+ML.Discretize.ByTiling(o
(Number IN [1,2]),4)+ML.Discretize.ByRounding(o(Number=4));
disc;

// Create instructions to be executed
inst := ML.Discretize.i_ByBucketing([3],4)+ML.Discretize.i_ByTiling([1,2],4)
+ML.Discretize.i_ByRounding([4]);

// Execute the instructions
done := ML.Discretize.Do(o,inst);
done;
```

Distribution (ML.Distribution)

This module exists to provide code to generate distribution tables and 'random' data for a particular distribution.

Each distribution is a 'module' that takes a collection of parameters and then implements a common interface. Other than the 'natural' mathematics of each distribution the implementation adds the notion of ranges (or NRanges). Essentially this means that the codomain (range) of the distribution is split into NRanges; this can be thought of as a degree of granularity. For discrete distributions this is naturally the number of results that can be produced. For continuous distributions you should think of the distribution curve as being approximated by NRanges straight lines. The maximum granularity currently supported in 1M ranges (after that the numeric pain of retaining precision is too nasty). The following routines are provided:

Routine	**Description**
Uniform (Low, High, NRanges.)	Specifies that any (continuous) value between Low and High is equally likely to occur.
StudentT (degrees-of-freedon, NRanges)	Specifies the degrees of freedom for a Student-T distribution. This module also exports InvDensity to provide the 't' value that gives a particular density value (can be useful as the 'tail' of a t distribution is often interesting).
Normal (Mean, Standard Deviation, NRanges)	Implements a normal distribution (bell curve), which shows mean 'mean' and standard deviation as specified, approximate by NRanges straight lines.
Exponential (Lamda, NRanges))	Implements the exponential (sometimes called negative exponential) distribution.
Binomial (p, NRanges)	Gives the distribution showing the chances of getting 'k' successful events in Nranges-1 trials where the chances of success in one trial is 'p'.
NegBinomial (p, failures, NRanges)	Gives the distribution showing the chances of getting 'k' successful events before 'failures' number of failures occurs (maximum total trials = nranges). The geometric distribution can be obtained by setting failures = 1.
Poisson (Lamda, NRanges)	For a poisson distribution the mean and variance are both provided by Lamda. It is a discrete distribution (will only produce integral values). Thus if NRanges is (say) 100 then the probability function for Poisson will be computed for values of 0 to NRanges-1.

The interface to the distribution provides:

```
  EXPORT t_FieldReal Density(t_FieldReal RH)
// The probability density function at point RH

EXPORT t_FieldReal Cumulative(t_FieldReal RH)
// The cumulative probability function from – infinity[1] up to RH

  EXPORT DensityV()
// A vector providing the probability density function at each range point –
   this is approximately equal to the 'distribution tables' that might be published
   in various books

EXPORT CumulativeV()
// A vector providing the cumulative probability density function at each range point
   – again roughly equal to 'cumulative distribution tables' as published in the back of
   statistics books

EXPORT Ntile(Pcnt
//provides the value from the underlying domain that corresponds to the given percentile.
  Thus .Ntile(99) gives the value beneath which 99% of all should observations will fall.
```

Most people do not really encounter StudentT as a distribution, rather they encounter the t-test. You can perform a t-test using the t distribution using the NTile capability. Thus the value for a single-tailed t-test with 3 degrees of freedom at the 99% confidence level can be obtained using:

```
a := ML.Distribution.StudentT(3,10000);
a.NTile(99); // Single tail
a.NTile(99.5); // Double tail
```

Note: The Cauchy distribution can be obtained by setting v = 1.

This module also exports:

```
GenData(NRecords,Distribution,FieldNumber)
```

This allows N records to be generated each with a given field-number and with random values distributed according to the specified distribution. If a 'single stream' of random numbers is required then FieldNumber may be set to 1. It is provided to allow 'random records' with multiple fields to be produced.

For example:

```
IMPORT * FROM ML;
//a := ML.Distribution.Normal(4,5,10000);
a := ML.Distribution.Poisson(40,100);
//a := ML.Distribution.Uniform(0,100,10000);
a.Cumulative(5);
choosen(a.DensityV(),1000);
choosen(a.CumulativeV(),1000);
b := ML.Distribution.GenData(200000,a);
ave(b,value);
variance(b,value)
```

The following performance timings were done generating 3 fields, two normal and one poisson:

Records	Time (secs)
50K	1.00
500K	2.29
5M	16.15
25M	80.2
50M	163
100M	316

FieldAggregates (ML.FieldAggregates)

This module works on the field elements provided. It performs the tasks shown below for ALL the fields at once. The following routines are provided:

Routine	**Description**
Simple stats	Mean, Variance, Standard Deviation, Max, Min, Count, Sums, etc
Medians	Provides the median elements.
Modes	Provides the modal value(s) of each field.
Cardinality	Provides the cardinality of each field.
Buckets/Bucket Ranges	Divides the domain of each field evenly and then counts the number of elements falling into each range. Can be used to graphically plot the distribution of a field
SimpleRanked	Lists the ranking of the elements from the smallest to the largest.
Ranked	Adjusts the simple ranking to allow for repeated elements (each repeated element gets a rank which is the mean of the ranks provided to each element individually)
NTiles/NTileRanges	Think of this as giving the percentile rank of each element, except you get to pick if it is percentiles (N=100) or some other gradation

Regression (ML.Regression)

The Regression module has 2 implementations. There is a sparse matrix implementation, ML.Regression.Sparse, and a dense matrix implementation, ML.Regression.Dense. Both of these are an implementation of Ordinary Least Squares regression using either LU or Cholesky factorization.

The regression attributes are:

Routine	**Description**
Regression.Sparse.OLS_Cholesky	Cholesky factorization of a sparse matrix for OLS regression.
Regression.Sparse.OLS_LU	LU factorization of a sparse matrix for OLS regression.
Regression.Dense.OLS_Cholesky	Cholesky of a dense matrix using the PB BLAS routines for OLS regression.
Regression.Dense.OLS_LU	LU factorization of a dense matrix using PB BLAS routines for an OLS regression.
Regress_Poly_X	Performs a polynomial regression for a single independent variable with a single dependent target.

The OLS routine can also return R^2 and Anova tables based upon the regression.

Currently, Poly handles polynomials up to X^3 and includes logarithms. Along with the other ML functions Poly is designed to work upon huge datasets; however it can be quite useful even on tiny ones. The following dataset captures the time taken for a particular ML routine to execute against a particular number of records. It then produces a table showing the expected running time for any number of records that are entered:

```
IMPORT ML;

R := RECORD
                INTEGER rid;
  INTEGER Recs;
                REAL Time;
                END;
d := DATASET([{1,50000,1.00},{2,500000,2.29},        {3,5000000,16.15},{4,25000000,80.2},
                                                      {5,50000000,163},{6,100000000,316},
                                                      {7,10,0.83},{8,1500000,5.63}],R);

ML.ToField(d,flds);

P := ML.Regression.Poly(flds(number=1),flds(number=2),4);
P.Beta;
P.RSquared
```

The R squared is a measure of goodness of fit.

Visualization Library (ML.VL)

The Visualization Library is a set of routines designed to easily transition raw tabular data into an assortment of visually intuitive charts. The scope of the project is not to provide the charts themselves, but rather a standardized interface that can be used to connect to existing, publicly available charting APIs, such as Google Charts and D3. Using this library, the user may produce charts of many different types without consideration for the details that are specific to any one of those libraries.

Using the library in its most rudimentary form can be as simple as this:

```
dFormatted:=VL.FormatData(dMyRawData,month);
VL.Chart(“MyLineChart”,dFormatted).Line;
VL.Chart(“ColumnChart02”,dFormatted).Column;
VL.Chart(“PieChart04”,dFormatted).Pie;
```

The above calls will produce three charts (Line, Column and Pie, respectively), which can be viewed in the ECL IDE directly as individual charts, or as a published service in a “dashboard” type of format. More advanced capabilities are also available that enable the user to specify options either globally or on a chart-by-chart level, such as the chart’s dimensions, title, etc.

Setting up

The assumption is being made that the ecl-ml repository has already been downloaded and placed in the user’s environment. The VL module is included in this suite of code. However, before the user can take advantage of the capabilities of the Visualization Library some changes need to be made to the environment so that the system knows how to create the charts. Specifically, the user needs to direct the compiler to a manifest file which contains the template information needed to translate the data into charts.

The manifest file is located in the subfolder “XSLT” under the VL module. To link the compiler to it, simply add the following line as a parameter to your compiler calls:

```
-manifest “[the location of ecl-ml]\VL\XSLT\manifest.xml”
```

In the IDE, the user can specify compiler arguments by following these steps:

1. On the IDE start-up screen, select **Preferences**.

2. Select the **Compiler** tab.

3. Enter the location of the manifest file (shown above) in the **Arguments** line.

Note: Compiler arguments are specific to the configuration, so if there are multiple configurations, ensure the proper one is selected. If the user wants to enable VL on all configurations, it will be necessary to place this option on each one individually.

Parameters

A call to a chart in the VL framework needs three parameters:

- A string containing the name to give the chart.

- A dataset containing the data to be presented, in the VL’s standard ChartData format.

- A Virtual Module containing the options to apply to the chart.

The first parameter is a simple string containing the name to give the chart. If more than one chart is called in a single job, it is important that this name is unique, or there will be a compiler error.

The second parameter is the data. In order to simplify the movement of data through the various levels of the VL, all data passed into it is expected it to be in a uniform format: VL.Types.ChartData. The macro “FormatData” is provided which will translate data into this format, where the user passes in the name of the incoming dataset and the name of the column that contains the X-Axis values. In the above example, the dataset name is dMyRawData, and the X-Axis labels are located in the column “month”.

The third parameter is a reference to the options that the user may adjust related to the position, size and other style-related attributes of the chart. This is a virtual module defined in VL.Styles. A default set of values is provided along with a few standard modifications that the user may desire (“Large”, “Small”, etc.). If the user does not specify the third parameter, VL.Styles.Default is assumed to be the options to apply.

Specifying an API Library

The VL.Chart attribute is a generic entry point the enables the user to be unconcerned about the underlying structure of the Visualization Library. It takes on the responsibility of deciding for the user which API library to call to produce the chart in question. In the case where a chart may exist in more than API library, a choice has already been made as to which one is considered the “default” when requesting a pie chart, for example.

There may be times, however, when a more advanced user want to have more control over which API library to call to produce a specific chart. In such a case, the user can call the interface to the API directly rather than through VL.Chart.

For example:

```
VL.Chart(“MyLineChart”,dFormatted).Line;
VL.Google(“MyLineChart”,dFormatted).Line;
```

These two lines produce the same results because Google is the default API for the Line chart type. But if the default were not Google, then the user could override that default by using the second line.

There is an interface-layer attribute for each API library with which the VL library interacts. The parameter list for each will be the same which enables the user to easily switch API’s when desired.

Using ML with documents (ML.Docs)

The ML.Docs module provides a number of routines which can be used to pre-process text and make it more suitable for further processing. The following routines are provided:

Routine	Sub-Routine	Description
Tokenize		Routines which turn raw text into a clean processing format.
	Enumerate	Applies record numbers to the text for later tracking.
	Clean	Removes lots of nasty punctuation and some other things such as possessives.
	Split	Turns the document into a token (or word) stream.
	Lexicon	Constructs a dictionary (with various statistics) on the underlying documents.
	ToO/FromO	Uses the lexicon to turn the word stream to and from an optimized token processing format.
Trans		Performs various data transformations on an optimized document stream.
	WordBag	Turns every document into a wordbag, by removing (and counting) multiple occurrences of a word within a document.
	WordsCounted	Annotates every word in a document with the total number of times that word occurs in the document and distributes tfdi information for further (faster) processing.
CoLocation		Functions related to the co-occurrence of NGrams and their constituents.
	AllNGrams	A breakdown of all n-grams within a set of raw document text, where n is specified by the user.
	NGrams	Aggregate information for every unique n-gram found within the document corpus.
	Support	The ratio of the number of documents which contain all of the n-grams in the parameter set compared to the corpus count.
	Confidence	The ratio of the number of documents which contain all of the n-grams in both parameter sets, compared to the number of documents which only contain the n-grams in the first set.
	Lift	The ratio of observed support of two sets of n-grams compared to the support if the n-grams were independent.
	Conviction	The ratio of how often one set of n-grams exists without a second set given that the two are independent.
	SubGrams	Compares the product of the document frequencies of all constituent unigrams to the n-gram they comprise.
	SplitCompare	Compares the document frequency of every n-gram to the individual frequencies of a member unigram and the remaining n-1-gram.
	ShowPhrase	If lexical substitution was using in constructing the AllNGrams dataset, this function will re-constitute the text associated with their corresponding numbers within an n-gram.
PorterStem		Standard Porter stemmer in ECL.
PorterStemC		Standard Porter stemmer in C (faster).

Typical usage of the Docs module

The following code demonstrates how the docs module may be used and show the result at each stage:

```
IMPORT ML;
IMPORT ML.Docs AS Docs;

d := DATASET([{'One of the wonderful things about tiggers is tiggers are wonderful things'},
                                  {'It is a little scary the drivel that enters ones mind
                                    when given the task of entering random text'},
                                  {'I almost quoted Oscar Wilde; but I considered that I
                                    had gotten a little too silly already!'},
                                  {'In Hertford, Hereford and Hampshire Hurricanes hardly
                                    ever happen'},
                                  {'It is a far, far better thing that I do, than I have
                                    ever done'}],{string r});

d1 := PROJECT(d,TRANSFORM(Docs.Types.Raw,SELF.Txt := LEFT.r));

d1;

d2 := Docs.Tokenize.Enumerate(d1);

d2;

d3 := Docs.Tokenize.Clean(d2);

d3;

d4 := Docs.Tokenize.Split(d3);

d4;

lex := Docs.Tokenize.Lexicon(d4);

lex;

o1 := Docs.Tokenize.ToO(d4,lex);
o1;

Docs.Trans(o1).WordBag;
Docs.Trans(o1).WordsCounted;

o2 := Docs.Tokenize.FromO(o1,lex);
o2;
```

Performance Statistics

The following performance statistics of these routines were observed using a 10 node cluster:

Routine	**Description**	**Result**	**Expected Order**
Clean and Split	22m documents producing 1.5B words.	60 minutes	Linear
Lexicon	1.5B words producing 6.4M entries.	40 minutes	NlgN
Lexicon	Creating 'working' entries from 1.5B words and the full 6.4m entry lexicon.	46 minutes	NLgN
Lexicon	Creating ‘working’ entries from 1.5B words and the ‘keyword’ lexicon, produces 140M words.	37 minutes	NLgN
Lexicon	Creating ‘working’ entries from 1.5B words and the ‘keyword MkII’ lexicon, produces 240M words.	40 minutes	NLgN
Lexicon	Creating ‘working’ entries from 1.5B words and the ‘keyword MkII’ lexicon, produces 240M words, using the ‘small lexicon’ feature.	7 minutes	Linear
Wordbag	Create WordBags from 140M words.	114 seconds	NlgN
Wordbag	Create WordBags from 240M->193M words.	297 seconds	NlgN

Useful routines for ML implementation

The following modules are provided to help with ML implementation:

- The Utility module
- The Matrix Library (Mat)
- The Dense Matrix Library (DMat)
- Parallel Block BLAS for ECL

Utility

The Utility module provides the following routines:

Routine	Description
ML.ToFieldElement/FromFieldElement	Translates in and out of the core field-element datamodel.
ML.Types.ToMatrix/FromMatrix	Translates from field elements to and from matrices.

The Matrix Library (Mat)

This library is in addition to and subservient to the ML library modules. The following routines are provided:

Routine	**Description**
Add	Add two matrices.
Decomp	Matrix Decomposition: LU, Choleski and QR.
Det	Determinant of a matrix.
Each	Some ‘element by element’ processing steps.
Eig	Decompose matrix into eigenvalue and eigenvector component matrices
Eq	Are two matrices equal.
Has	Various matrix properties.
Identity	Construct an Identity Matrix.
InsertColumn	Add a column to a matrix.
Inv	Invert a matrix.
Is	Boolean tests for certain matrix types (Identity, Zero, Diagonal, Triangular etc).
Lanczos	Lanczos decomposition
MU	Matrix Universe. A number of routines to allow multiple matrices to exist within the same ‘file’ (or dataflow). Useful for iterating around loops etc.
Mul	Multiply two matrices.
Pca	Principal Component Analysis
Pow	Multiplies a matrix by itself N-1 * (and demo’s MU).
Repmat	Construct a matrix consisting of M-by-N tiling copies of the original matrix
RoundDelta	Round all the elements of a matrix if they are within delta of an integer.
Scale	Multiply a matrix by a constant.
Sub	Subtract one matrix from another.
Substitute	Construct a matrix which is all the elements of the right + any elements from the left which are not in the right.
Svd	Singular Value Decomposition
Thin	Make sure a matrix is fully sparse.
Trans	Construct the transpose of a matrix.
Vec	A vector library, to create vectors from matrices and assign vectors into matrices.

The following examples demonstrate some of these routines:

```
IMPORT ML;
IMPORT ML.Mat AS Mat;
d := dataset([{1,1,1.0},{1,2,2.0},{2,1,3.0},{2,2,4.0}],Mat.Types.Element);

Mat.Sub( Mat.Scale(d,10.0), d );
Mat.Mul(d,d);
Mat.Trans(d);
```

The Dense Matrix Library (DMat)

This module provides the following routines:

Routine	**Description**
Add	Sums two matrices.
Converted	Module for several different conversions.
Decomp	Decompose a matrix.
Identity	Create an identity matrix.
Mul	Multiply two matrices.
Scale	Multiplies a matrix by a scalar.
Sub	Subtracts two matrices.
Trans	Matrix transpose

These routines are intended to be similar to the Mat module.

Parallel Block BLAS for ECL

The ML Libraries provide an implementation of selected BLAS operations for use in a Parallel Block matrix application. The BLAS operations work on double precision floating point numbers. The functions provided are: matrix and matrix multiply and sum; scalar and matrix multiply and sum; scale vector; symmetric rank update; triangular matrix solver; Cholesky factorization; and LU factorization.

To use the Parallel Block BLAS routines, the user must decide on the partitioning for the matrices. The PBblas.Matrix_Map attribute is used to communicate the matrix and partition dimensions to the BLAS attributes. It is the responsibility of the user to employ compatible partitioning schemes.

The PBblas module provides the following routines:

Routine	**Description**
MakeR8Set	Packs a dataset of cells for a partition into a set of double precision numbers.
Matrix_Map	Specifies the dimensions of a matrix and the dimensions of the partitions.
PB_daxpy	Multiplies vector X by a scalar and sums vector Y.
PB_dbvmm	Efficient matrix multiply when the result is a single partition. Multiplies the product of matrices A and B with a scalar alpha, and then sums with matrix C multiplied by a scalar beta.
PB_dbvrk	Matrix update with a block vector.
PB_dgemm	General purpose matrix multiply. Multiplies a the product of matrices A and B with a scalar alpha, and then sums with matrix C multiplied by a scalar beta.
PB_dgetrf	LU Factorization.
PB_dpotrf	Cholesky factorization.
PB_dscal	Scalar multiply.
PB_dtran	Matrix transpose.
PB_dtrsm	Solves either Ax = B or xA = B.
PB_Extract_Tri	Extracts the upper or lower triangular. Useful for extracting the L and U matrices from the composite matrix created by the LU factorization.

Installation notes

The ML Libraries can use the BLAS libraries to speed up certain matrix operations. You may install BLAS (Basic Linear Algebra System) or ATLAS, or some other package containing a BLAS. The developer version should be installed.

The Attributes that employ BLAS require the cblas.h header file and the cblas library. These files must be located in directories specified in default search paths for headers and libraries. In some systems, you may need to use a symbolic link. For example, in CentOS the header file is installed in a usual place but the library is installed in /usr/lib64 which is not where the linker looks by default.