Model Exporter Reference Manual

Contents

Introduction

Model_Exporter converts models created within the PLS_Toolbox or Solo chemometrics modeling environments into an interpretable format for use outside of these products. Models exported to Matlab or Python produce a stand-alone "predictor" script which includes a simple-to-implement formula to perform a model prediction. Models exported to XML can be used with the included C# or Java interpreters or with a user-supplied interpreter to make predictions on new data. Model_Exporter takes as input a standard model structure created in PLS_Toolbox or Solo and outputs the model into one of three formats: an XML file (executable by a user-supplied external parser or the Java or C# Model Exporter Interpreter class provided with Model_Exporter), an m-file (executable in MATLAB – separately distributed by Mathworks, Inc – without any additional toolboxes, or LabView with their MathScript addon package) or a Python file (executable in a Python 3.6 or higher environment).

The exported models require very few resources to be executed. Specifically, they requires floating-point numerical calculations, a small amount of memory, and the overhead resources required by the specific interpreter or language environment.

This documentation describes the use of the Model_Exporter, the use of exported M-file and Python file formats as well as to help in the design of external XML parsing engines. Model_Exporter includes a freely-distributable interpreter class with versions in C# and Java as described on the Model Exporter Interpreter page. In addition, an example interpreter engine is supplied for the PHP language (often used for web-page scripting predictions; see http://www.php.net for more information on PHP). Additional engines may be available - Contact Eigenvector Research, Inc. (mailto:helpdesk@eigenvector.com) for more information. Latest version release notes can be found at http://wiki.eigenvector.com/index.php?title=Model_Exporter_Release_Notes

System Requirements

Model_Exporter can be executed from either the MATLAB computational environment (Mathworks, Inc., Natick, MA (http://mathworks.com)), or Solo (Eigenvector Research, Inc., Wenatchee, WA). Model_Exporter converts models created by PLS_Toolbox 3.5 or higher or Solo 4.0 or higher.

Matlab-Based Exporter Requirements

For execution of Model_Exporter within the MATLAB environment, the following is required:

Compatible with any version of Matlab released within five years of this product's release. For example, Model exporter 3.3.0 released in 2016, is guaranteed to be compatible with any version of Matlab released within 5 years of the 2016 release data, so in this case Matlab 2011 would be the oldest version of Matlab that we fully guarantee compatibility.
256 MB RAM (recommended – less may be required)

Solo-Based Exporter Requirements

For execution of the Model_Exporter, the following is recommended

Solo+Model_Exporter 4.1 or higher
Operating system requirements as listed for the specified Solo version
200 MB Disk Space (for installation; some models may require additional space)
256 MB RAM (recommended – less may be required)

Requirements for Using Exported Models

The requirements to execute an exported model vary depending on the interpreter used, the number of variables in the modeled data, and the complexity of the model (i.e. the number of factors/components included in the model and the types of preprocessing used).

Memory requirements depend on the precision required for the application, the number of variables in the data and the total number of factors in the model. For example, a model working on 10,000 variables and 5 factors would require around 1MB for double-precision calculations and 500KB single-precision calculations.

The software which executes the specific file formats may have additional requirements. See the file format description sections later in this manual for where to locate model execution details.

Installation

Model_Exporter is installed by adding the Model_Exporter folder to your Matlab path. Once installed, PLS_Toolbox will recognize the installation and enable the proper menu items. Solo+Model_Exporter has the folder pre-installed.

For Matlab:

1. Make the Model_Exporter folder your Current Folder in Matlab.
2. Run the command 'setpath' from the command line.

Supported Methods

Model_Exporter supports the following model types:

PCA – Principal Components Analysis model
PLS – Partial Least Squares regression model
PLSDA – Partial Least Squares discriminant analysis model
PCR – Principal Components Regression model
CLS – Classical Least Squares Regression model
SVM – Support Vector Machine Regression model
SVMDA – Support Vector Machine Classification model
ANN – Artificial Neural Network Regression model using BPN algorithm
ANNDL - Artificial Neural Network Deep Learning Regression model
MLR - Multivariate Linear Regression model
LWR - Locally Weighted Regression*

and preprocessing methods:

Absolute value
Arithmetic
Autoscale
Baseline (specified)

- Derivative (SavGol)
- Detrend
- ELS
- EPO
- GLS weighting
- Log Decay Scaling
- Log10
- MSC (median, windows, and subinds not currently supported)
- Mean center
- Median center
- Normalize
- OSC
- Pareto Scaling
- Poisson Scaling
- SNV
- Smooth (SavGol)
- Sqrt Mean Scale
- Transmission to Absorbance
- Variance Scaling
- Weighted Least Squares Baseline Correction*
- Whittaker Baseline Correction*

*requires the use of the evriMEaddon module

CLS algorithm options 'nnls' and 'cnnls' are only supported for Matlab target

Normalization and Baseline support windowing. Normalization supports type 1 (area) and type 2 (length) normalization, but does not support 'Inf' type normalization.

Arithmetic does not support using individual variables.

Model_Exporter does not support replacement of missing values (values must be measured for all variables).

Validity of Exported Model Predictions

Exported models have been validated against "native" PLS_Toolbox/Solo model predictions for all supported analysis and preprocessing methods, with one exception. The exception is the case where the model was built using calibration X-block which has excluded variables and Savgol (smoothing or derivative) preprocessing is used. The one situation where exact agreement may NOT occur requires the following three conditions for the exported model:

1. The model is built from X-block data which has excluded variables
2. Savgol preprocessing with option "useexcluded"= true is the second or later preprocessing step
3. The Savgol preprocessing step must be preceded by one of the following preprocessing methods: 'arithmetic', 'Autoscale', 'simple baseline', 'derivative', 'EpO', 'GLS Weighting', 'logdecay','Normalize', 'sqmnsc', 'smooth', or 'detrend'

The cause of this exceptional case is related to exported models using only included variables, as this affects the ability of smoothing or derivative (Savgol) preprocessing steps to correctly process variables near excluded variables.

If a model with the conditions above is exported then a warning dialog will alert the user of the potential problem. The user can decide to cancel the exporting or not. In the latter case the user should test the exported model using typical samples to see if the prediction behavior is acceptable.

If Savgol preprocessing is used then it is best, if possible, to include it as the first preprocessing step, since this usage will be valid.

Exporting a Model

Exporting from PLS_Toolbox and MATLAB

Model_Exporter is easily called from the MATLAB environment. After adding the Model_Exporter folder to the MATLAB path, a model can be exported by simply calling the exportmodel function, passing the model structure itself, and an optional input specifying the file name and type to which the exported model should be written. When filename is omitted, Model_Exporter will prompt for a filename, file type, and location.

```
exportmodel(model,filename)
```

```
exportmodel(model,filename,options)
```

The third parameter, options, allows specification of how excluded variables are handled, how numerical values are stored (text or binary), and whether the exported m-file is a script or function. See below for further details of options.

Model_Exporter is also accessible from the PLS_Toolbox through the Analysis GUI. With the model to export loaded into the Analysis GUI, go to the **File** > **Export Model** > **To Predictor...** menu and select the file type to export from the flyout menu.

Exporting from Solo

When installed with the stand-alone Solo software, a model is exported from the Analyis GUI. With the model to export loaded into the Analysis GUI, Go to the **File** > **Export Model** > **To Predictor...** menu and select the file type to export from the flyout menu.

Handling Excluded Variables

When excluded variables are detected within a model, the user will be given two options for how to handle these variables.

1. Compress Model – Model_Exporter will attempt to remove all references to excluded variables. The created predictor will expect values for only the included variables.
2. Use Placeholders – Model_Exporter will create a predictor which expects values for all variables, excluded or included, although excluded values will be ignored.

The choice between these two methods depends on the environment in which the exported model is going to be used. If it is easier to always provide all variables to the predictor, then the “Use Placeholders” option is probably preferred. If, instead, only the included variables will be available (e.g. the excluded variables are not going to be measured), compressing the model is the correct approach.

In general, the two methods give identical numerical results with the sole exception of models which make use of smoothing and derivative preprocessing. These methods may give slightly different “edge effects” after compressing a model and validation of such models is encouraged.

In either case, the header information in the exported model will always reflect the number of variables expected and any labels or axisscale information for those variables.

Storing Numerical Values as Binary

With large numbers of variables, and with certain types of preprocessing (e.g. derivatives and smoothing), the numerical matrices needed to apply the model can become quite large, particularly when stored in the standard text format of an exported model. When the output target is either Matlab (.m) or Python (.py), you have the choice to store the numerical values in one of three formats:

- Text in the script (Default)
- Binary data file in DOUBLE (64-bit) precision
- Binary data file in SINGLE (32-bit) precision

Text in the script is the default format to store numerical values and allows all the model information to be included in a single file (the text script.) The second two options instead store these values in a separate binary file as a simple stream of numerical values of the indicated precision. When the binary formats are selected, the script is written to automatically open the binary file and read in the values from there instead of parsing them out of the script.

Notes:

1. This file format is currently only available for scripts exported in the m-file format. Eigenvector Research (mailto:helpdesk@eigenvector.com%7CContact) if you have interest in using a similar format for other export formats.
2. Single Precision Binary will reduce the accuracy of the predictions due to rounding error. The extent of error will depend greatly on the noise level of the data and the precision required by the model. Models exported with this precision should be validated with known samples to determine the effect of rounding on the predictions for the given model.
3. It is assumed that the binary file is in the "current working directory" (unless the script is edited to change the file location.)

M-file Format

The m-files output by Model Exporter are stand-alone. That is, they can be run by the MATLAB computational environment (available from Mathworks, Inc., http://www.mathworks.com) without any additional toolboxes or the LabVIEW environment (available from National Instruments, Inc., http://www.ni.com) with any MathScript-enabled package.

For maximum flexibility, an exported model is written as a script which expects only to find a variable named x in its workspace. This variable provides the input data to which the model should be applied. It is important to note that the variable x will be modified by the script and, thus, the caller should not expect the variable to remain unchanged. See "Creating Functions from Exported Models", below, for more information on how to isolate the script and call it as a function. (Those unfamiliar with MATLAB scripts and functions should read the MATLAB documentation describing these concepts and the associated "variable scope" documentation.)

The input variable x should be a vector, representing a single sample, and the output will be a prediction for this one sample.

Options

When exporting from the Matlab environment to a .m file using the *options* parameter it is possible to specify preferred behavior for these choices. The available options are:

- **handleexcludes**: [{'ask'} | 'ignore' | 'placeholders'} Governs how excluded variables should be handled.

  'ignore' = attempt to remove all references to excluded variables. Only included values will be expected.
  'placeholders' = expect values for all variables, although excluded values will not be used by model.
  'ask' = prompt user for desired behavior.

- **datastorageformat**: [{'ask'} | 'text' | 'binarydouble' | 'binarysingle'] Governs output format of numerical values.

  'text' = store numerical values as text in the script (the normal output mode).
  'binarydouble' = store as binary data file in DOUBLE precision.
  'binarysingle' = store as binary data file in SINGLE precision.
  'ask' = prompt user for desired format.

Note: Single Precision Binary will reduce the accuracy of the predictions due to rounding error. Validate results against known samples if single precision is used. Note: Binary output formats provide for smaller memory footprint but require parsers that can execute binary file read instructions.

- **creatematlabfunction**: [{'no'} | 'yes'] Governs m-file format specifying whether to create a Matlab script or function.

  'yes' outputs m-files with appropriate code to allow calls to the model application in a functional form.
  'no' outputs m-file in script form.

- **distancemetric**: ['yes' | {'no'}] Governs inclusion of distance metric which is an approximation of the nearest neighbor distance in score space (relative to the calibration samples.) The returned value will be normalized such that a value of 1 (one) indicates the given sample is as far from its nearest neighbor as was seen for any calibration sample. A value greater than 1 indicates a sample which is likely unusually distant from any calibration sample and may be an "inlier".

Input Data

The expected length (number of elements) and contents of the input x vector are defined in the comments and initial sections of the exported model script. The script, as exported, does not use this information to perform any validity testing on the input variable. This information is only provided to indicate to the user what type of data is expected.

The example below shows the part of an exported model which indicates the expected data size and associated context information. This particular model expects input data of ten variables as a row vector (as described by inputdata.size).

```
%     .size = [ 75 10 ];
%     .include_size = [ 75 10 ];
```

The user can make use of this information to assure the data being passed to the model is correct. Again, as written, the script provides no testing. Incorrect data sizes will be indicated by a runtime error when executing the script.

Returned Results

The results available from a model prediction will be present as variables in the script's workspace. The user is responsible for making use of these variables as needed. The following list specifies the supported results which may be of interest to the user.

> **scores** - Scores for each component as a row vector.
> **T2** - The Hotelling's T^2 as a scalar value.
> **Q** - The sum squared x residuals (Q value) as a scalar value.
> **Tcon** - Variable contributions to T2 as a row vector.
> **Qcon** - Q residuals contributions (x residuals) as a row vector.
> **x** - The preprocessed version of the input data.
> **Xhat** - Model estimate of the data as a row vector (in preprocessed units - comparable to the preprocessed **x**).

Note that returned Q and T2 are their "reduced" form, divided by the model's confidence limit value for Q or T2 (95 percent by default). A value of Q or T2 greater than 1 indicates an unusual sample at the model's confidence limit level for Q or T2.

All regression and PLSDA models return the following additional value:

> **yhat** - Model prediction for y (predicted y value) as a scalar value or vector.

PLSDA discriminant analysis models also return an additional value

> **probs** - Model predicted probability of the input sample belonging to each class, where the classes are ordered as unique(y), as a vector. (y refers to the classes variable originally used in building the model).

SVM regression analysis models return values:

> **yhat** - Model prediction for y (predicted y value) as a scalar.
> **nsvs** - Number of support vectors used by the model, as a scalar.

SVMDA discriminant analysis (classification) models return values:

> **probs** - Model predicted "probability" of the input sample belonging to each class, where the classes are ordered as shown in classorder (below). Note that the probability reported here is **not** the same as the probability reported by the SVM algorithm (based on a maximum likelihood calculation). Instead, this is based on the classvotes reported below. The class votes are normalized to give fraction of votes for each class. This fraction is raised to the power of 10, then normalized to unit area again. This gives a ROUGH estimate of probability where the class with the highest votes also gets the highest probability and the remaining classes are ranked in decreasing order. The log of the probability is roughly proportional to the number of class votes that would have to change to cause the assignment to change.
>
> > NOTE: For historical reasons, the output **prob** will also contain the identical probabilities as **probs**.

classvotes - Votes cast in favor of each class, as a vector. The class with most votes is the predicted class of the input sample.
classorder - Is a vector of class numbers identifying which class each classvotes value is associated with. For example, if the second entry in classvotes (or probs) has the largest value then the second value in classorder give the winning class number. See the model's model.classification.classnums and model.classification.classids to translate between class numbers with class names. Ties between two or more classes are resolved by choosing the first.
nsvs - Number of support vectors used by the model, as a scalar.
df - Vector of decision function values for pairwise classifiers, as a vector. If there are N classes then there are N*(N-1)/2 pairwise classifiers used. The decision functions are in order: class 1-2, 1-3,...1-N, 2-3, 2-4, ...,2-N,..., (N-1)-N. The classvotes are based on the decision function values.

Note these exported model results should be the same as results from SVMDA when using option probabilityestimates = 0 (even if the exported model was built using option probabilityestimates = 1). Thus the exported model's predictions should only be validated against SVMDA models built using probabilityestimates = 0.

ANN regression analysis models return values:

yhat - Model prediction for y (predicted y value) as a scalar.
anntype - ANN algorithm 1 = 'BPN', 2 = 'Encog'.

Creating Functions from Exported Models

Although the exported model is written as a script which would normally operate in the base workspace of MATLAB, the user can also wrap the script into a function by simply adding a standard function definition to the script file. A function wrapper keeps the input variable x from being modified outside the function. This approach tends to be safer than a script, but is not implemented by default in order to provide the widest flexibility to the user.

An example function line is provided in the exported model file (commented out) along with instructions for customization. In addition, there is an example block of code (also commented out by default) which will return “expected information” about x if the function is called without any inputs.

In general, the function definition requires only one input, x, and can output any of the variables which are present after the script's execution. An example would be:

```
function [scores,Q,T2,Qcon,Tcon] = mymodel(x)
```

This function definition returns the vectors: scores, Qcon, and Tcon, as well as the scalar values: Q and T2 to the caller.

Note, as discussed above, the user can have this conversion of the exported m-file from a script to a function applied automatically by specifying the exportmodel option **creatematlabfunction** = 'yes'.

Python File Format

The Python file output by Model Exporter can be run in a Python version 3.6 or higher environment. It requires the NumPy (http://www.numpy.org/) package be available.

For maximum flexibility, an exported model is written as a Python module (.py file) which expects only to find a NumPy array named x in its variable space. This variable provides the input data to which the model should be applied. It is important to note that the variable x will be modified by the script and, thus, the caller should not expect the variable to remain unchanged.

The input variable x should be a 1xn or mxn array, representing a single sample or multiple samples, and the output will be a prediction(s).

Input Data

The expected length (number of elements) and contents of the input x vector are defined in the comments and initial sections of the exported model script. The script, as exported, does not use this information to perform any validity testing on the input variable. This information is only provided to indicate to the user what type of data is expected.

The example below shows the part of an exported model which indicates the expected data size and associated context information. This particular model expects input data of ten variables as a row vector.

```
#   .size = [ 75 10 ];
#   .include_size = [ 75 10 ];
```

The user can make use of this information assure the data being passed to the model is correct. Again, such testing is not provided by the script as written. Incorrect data sizes will be indicated by a runtime error when executing the script.

Using Exported Models

The exported model is written as a Python module with the user's specified name, for example 'pcaexample.py'. This module can be imported and used in Python code.

All exported model Python modules contain an "apply" function which requires only one input, x, and returns a dictionary containing results. The input, x, should be a NumPy array of test samples with shape 1xn or mxn for single or multiple sample data. An example would be:

```
# PCA model was exported to Python file named 'pcaexample.py'
import pcaexample

# Use absolute or relative path
filename = '../arch_5row.csv'

x = np.loadtxt(open(filename, "rb"), delimiter=",", skiprows=0)

# Apply exported model
res = pcaexample.apply(x)

# This PCA example function returns a dictionary containing the scores, loads, Qcon, and Tcon,
# as well as the scalar values Q and T2 to the caller.   # Report scores
print('PCA: [\'scores\']: \n', res['scores'])
```

Returned Results

The particular results returned from applying the exported model depend on the model type and are the same as those listed under the M-file format description. These will be contained in a returned dictionary 'res' and can be accessed as in this example:

```
print('PCA: [\'scores\']: \n', res['scores'])
```

The full list of returned quantities is available from the dictionary keys:

```
# Query the resulting dictionary for keys
print('\nPCA: res keys: ', res.keys())
```

XML File Format

The XML files output by Model Exporter are not directly usable. They can be used in any programming language for which an interpreter is available to translate the model's XML representation into a form which can be used by that programming language. The Model Exporter software package includes two such interpreters, one for .NET languages and one for Java. For more details see Model_Interpreter (http://wiki.eigenvector.com/index.php?title=Model_Exporter_Interpreter). The input variable x should be a vector, representing a single sample, and the output will be a prediction for this one sample.

Numerical Matrix Definitions

The XML format utilizes custom tags to define various parts of the model. For some tags, the content is a vector or matrix of values. In these cases, a comma character delineates different column elements and semicolon indicates the end of a matrix row and the beginning of the next. All white space is ignored. If a given matrix contains only one row, it is described as a "row vector". A matrix with a single column is described as a "column vector". Orientation of such vectors is critical to the mathematical operations and must be parsed appropriately.

XML Structure

The XML file will consist of a top level <model> tag which will contain an <information> tag, an <inputdata> tag, and one or more step segments, each wrapped in a separate <step> tag.

<model>

- **<information>** General information on the encoded model.
  - **<source>** Text description of file source (EVRI Model_Exporter).
  - **<modeltype>** Standard model method acronym (PCA, PLS, etc).
  - **<description>** Text description of model including preprocessing, data size(s), and number of components. Each row of this multi-row string is delineated by <sr> (string row) tags.
  - **<datasource>** Information block of modeled calibration data. <datasource> is a multi-cell table format. There will be one column of information for each block of data required by the given modeltype (e.g. PCA requires 1 block, PLS requires 2). Each <td> tag will contain a number of sub-fields describing the data used for the given block. Informational only, sub-fields may change.
  - **<outputs>** Table-formatted (TR and TD wrapped) array of the names associated with the columns of the yhat output (if any). These can be used by the caller to assign string descriptions to each output. They are not used by the interpreter itself, however.

</information>

<inputdata> Specific requirements for input data including the following information:

> **<size>** Numeric class row vector describing the size expected for the input data (x). The first element of the vector gives the expected number of rows, the second is the expected number of columns.
>
> **<axisscale>** Numeric class row vector providing the expected axisscale of the input values. The actual values stored in the axisscale vector are completely dependent on the application and the analytical method used and may be empty.
>
> **<label>** Strings (delimited by <sr> sub-tags) defining the names of the variables expected in the input data (x). The names are dependent on the application and the analytical method used and may be empty.

</inputdata>

<step> Repeated tag for each step required for making a prediction using this model. Will contain the following sub-fields:

> **<sequence>** Numeric class single value indicating the order in which this step should be performed. The steps are generally included in the XML file in sequence-order (sequence 1 will be the first step in the file), but this field can be used to assure in-order processing of steps.
>
> **<description>** String class description of the step (informational only)
>
> **<constants>** Contains information on constants required by this step. Each constant is defined as a sub-tag herein. The name of the constant is the sub-tag name and will contain a matrix (or vector) of values to use for the given constant. See below for more information.
>
> **<script>** One or more rows of strings describing the mathematical operations to perform this step. When more than one mathematical operation is to be performed, each will be given in a separate string row <sr> tag. However, these can be ignored. Each mathematical operation will be terminated with a semicolon.

</step>

(Additional <step> tags located here...)

</model>

See the provided files "pcaexample.xml" and "plsexample.xml" for full examples of the XML structure.

Using XML-Exported Models

TODO: describe briefly the .NET and Java ModelInterpreters and refer to the examples

Using XML-Exported Models in .Net

TODO:

Using XML-Exported Models in Java

TODO:

Requirements for XML Interpreters

To execute each of the <step> segments contained in the XML file, an interpreter must be able to parse the constants defined into matrices and be able to execute the script commands. The following give the specifications for an interpreter.

For examples of interpreters, see the Model_Exporter Interpreter objects in folder: interpreters/MEInterpreter, or the PHP interpreter in interpreters/predict.php. These are all distributed with Model_Exporter and are *freely-distributable without additional licensing*.

Managing of Constants and Variables

- The interpreter must maintain a "workspace" of stored constants and variables in which the matrices can be accessed by a variable name (specified by the tag in which the given constant was read, for example:

```
<s class="numeric" size="[1,1]">4</s>
```

would define a constant "s" which was equal to the scalar value 4).

- Constants are NOT case sensitive and any interpreter must be written to consider the upper or lower case variables as the same.
- "Constants" are just pre-defined variables. Although every effort will be made to avoid changing these values, it is NOT a rule that these "constants" cannot be changed – scripts may modify and overwrite these values. They are called "constants" because they are initially defined by the model.
- The enclosing tag for the constant will define the class of the constant (in this application, constants will always be "numeric") and will also define the size of the constant using the attribute 'size'. For example,

```
<s class="numeric" size="[1,5]">
```

defines that the enclosed constant will be a row vector (1 row) of 5 elements (5 columns).
- Prior to the execution of the script(s), the XML interpreter must place a variable named "x" (lower-case) in their workspace. This variable must contain the data to which the model should be applied. The value of "x" will be modified by the script so, following initial assignment, no alteration of this variable should be done outside what is specified by the script.
- All constants/variables must be retained for the entirety of a given step. In many cases, the variables remaining in the workspace will contain results of interest to the caller and, therefore, all workspace values should be retained. The variable "x" must always be present.

Script Execution

The following lists define the script commands which must be supported by the interpreter (scripts may contain only these commands). When applicable, the Matlab operator corresponding to the given function is given. Interpreters do not need to interpret these operators. They will never be used in any script and are provided here only for reference.

Single Input Functions

```
C = function(A);
 abs             Absolute Value     Removal of sign of elements ( abs(A) )
 log10           log (base 10)      Base 10 logarithm of elements ( log10(A) )
 transpose       transpose array    Exchange rows for columns ( A' )
```

Double Input Functions

```
C = function(A,B);
  plus         Plus                            Addition of paired elements ( A+B )
  minus        Minus                           Subtraction of paired elements ( A-B )
  mtimes       Matrix multiply (dot product)   Dot product of matrices ( A*B )
  times        Array multiply                  Multiplication of paired elements ( A.*B )
  power        Array power                     Exponent using paired elements ( A.^B )
  rdivide      Right array divide              Division of paired elements ( A./B )
  cols         Index into columns of matrix    Select or replicate columns  ( A(:,B) )
  rows         Index into rows of matrix       Select or replicate rows     ( A(B,:) )
```

Mathematical Operation Requirements

- All mathematical operations are expected to be performed using signed, single precision numbers.
- With the exception of mtimes (dot product), all operations are "element-by-element". That is, the two matrices passed will be equal in size (see scalar exception below) and the mathematical operation is performed between each element of matrix A and its corresponding element in matrix B. The output matrix C is always the same size as A and B.
- Scalar Exception (except mtimes): A or B may be a scalar even if the other isn't. In this situation, the scalar input must be interpreted as an appropriately-sized matrix containing all the same value.
- mtimes (dot product) is performed using the standard linear-algebraic dot-product operation. In generic terms, the input matrix A will contain m rows and k columns, the input matrix B will contain k rows and n columns and the output matrix C will contain m rows and n columns. The following equation is used to calculate each element of the C matrix (loop for i = 1 to m and for j = 1 to n):

$$C_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + A_{i,3}B_{3,j} + \ldots + A_{i,k}B_{k,j}$$

  Subscripts indicate the row and column indexing (respectively) into the given array. When either A or B is a scalar, the mtimes operation should be handled as a "times" operation. That is, the operation becomes an element-by-element multiplication where each element of the matrix input is multiplied by the scalar value and C is the same size as the input matrix.

- cols and rows indexing operations should expect a row vector for B that may have repeated elements (which allows replication of a given row or column). For example, given a row vector for B of

  B = [1 1 1 2 2 2]

  passed into the cols operation, this would replicate column 1 three times then replicate column 2 three times giving a total of 6 columns in the output.

Script Execution Requirements

- The format for a single script command is:

  ```
  C = function(A,B);
  ```

  where function is one of the above functions, A and B are the pre-defined constants / variables to use as input to function, and C is the output. Input B will be omitted for functions which require only one input. Each command of the script will end in a semi-colon ";". All commands must be performed in the order in which they appear in the script.
- The expected size, axisscale, and labels associated with x will be stored in the <sourcedata> tab (if any exist). These values can be used by an XML interpreter to verify the data being analyzed.
- Constants are NOT case sensitive and any interpreter must be written to consider the upper or lower case variables as the same.

Returned Results

The results returned by a model prediction will be present as variables in the interpreter's workspace upon completion of the XML parsing. The returned results are the same as those listed for the M-file format.

Requirements for XML Writers

The following rules are to be followed by the script creation algorithm of Model_Exporter. These rules may be of interest to script interpreters, but should not have any critical impact on interpreter design.

- Nesting of functions is not allowed. Functions can only take variables or pre-defined constants as input.
- NO iterative processes are supported. All scripts must be straight-through executing (no control structures such as "ifs", "while", etc are supported.)
- Missing data replacement is not supported.
- As of version 1.0 of this product, only variables or pre-defined constants may be used in a function. No "in-line" constants may be used. For example:

```
 C = minus(A,1);
```

is invalid because the constant "1" has to be pre-defined. This command should instead be written where the "1" is pre-defined as a constant and the name of that constant is used.

- Variables are NOT case sensitive and any interpreter must be written to consider the upper or lower case variables as the same. Note, however, that the Matlab output of this function will be case-sensitive code so the scripts should try to be consistent in case, even if other interpreters won't care.

evriMEaddon

The original intent of Model_Exporter was to produce lean code, often for low resource targets, where a minimal number of basic matrix operations are needed to convert application of a model into a top-down recipe. As such, some model and preprocessing methods are precluded from export in Model_Exporter due to requiring high level functions, iterations, and the like. In response to customer requests to broaden this list, especially in instances where computing resources are not so limited, we have added to the Model_Exporter architecture an add-on library named

```
evriMEaddon
```

(MATLAB and Python platforms). The main code exported by Model_Exporter will continue to consist of basic matrix operations and be strictly top-down, but additional functionality will be available through a call to this library.

Enable Support

- Weighted least squares baseline correction
- Whittaker baseline correction

Examples

For example, in an exported model calling for Whittaker baseline correction, the following code is generated defining the parameters for the preprocessing algorithm (following examples in MATLAB code):

```
whittakerOPS.lambda = 100;
whittakerOPS.p = 0.001;
whittakerOPS.maxiter = 100;
whittakerOPS.maxtime = 600;
```

The baseline correction is performed with the following command:

```
x = evriMEaddon('evriMEwhittaker', x, whittakerOPS);
```

The syntax will always consist of calling the library

```
evriMEaddon
```

and the argument list consists of

- the name of the module
- the variable containing the data
- any additional parameters (for example, weight least squares baseline correction uses the axis scale from mode 2 of the data)
- an options `structure` (MATLAB)/`dictionary` (Python) containing the necessary parameters for applying the algorithm

Note that these commands will be generated automatically by Model_Exporter, so no further modification is needed on the part of the user.

The files `evriMEaddon.m` and `evriMEaddon.py` may be found in the folder "evrimeaddon" under the main installation folder for Model_Exporter. During deployment, these files need to be accessible within the environment of the exported model code. In the case of Python, the `evriMEaddon` function needs to be imported within the application via

```
from evriMEaddon import evriMEaddon
```

or add `evriMEaddon.py` to the Python path.

Retrieved from "https://www.wiki.eigenvector.com/index.php?title=Model_Exporter_Reference_Manual&oldid=11635"

This page was last edited on 9 August 2022, at 08:23.