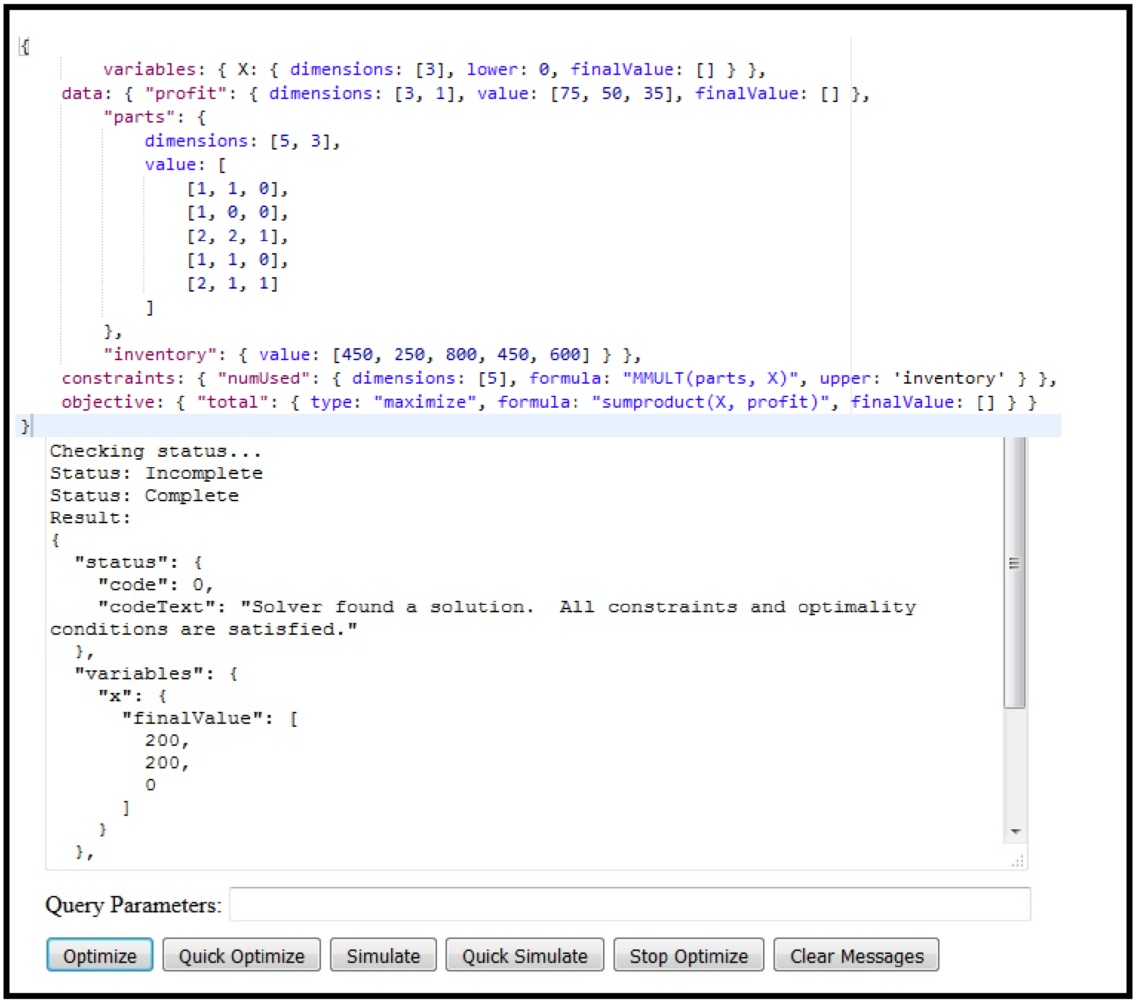
RASON Modeling Language Reference Guide



```
{
    variables: { X: { dimensions: [3], lower: 0, finalValue: [] } },
  data: { "profit": { dimensions: [3, 1], value: [75, 50, 35], finalValue: [] },
      "parts": {
          dimensions: [5, 3],
          value: [
              [1, 1, 0],
              [1, 0, 0],
              [2, 2, 1],
              [1, 1, 0],
              [2, 1, 1]
          ]
      },
      "inventory": { value: [450, 250, 800, 450, 600] } },
  constraints: { "numUsed": { dimensions: [5], formula: "MMULT(parts, X)", upper: 'inventory' } },
  objective: { "total": { type: "maximize", formula: "sumproduct(X, profit)", finalValue: [] } }
}
```

```
Checking status...
Status: Incomplete
Status: Complete
Result:
{
  "status": {
    "code": 0,
    "codeText": "Solver found a solution.  All constraints and optimality
conditions are satisfied."
  },
  "variables": {
    "x": {
      "finalValue": [
        200,
        200,
        0
      ]
    }
  },
```

Query Parameters:

Copyright

Trademark

Patent Pending

Systems and Methods for Automated Risk Analysis of Machine Learning Models.

Contact Us

Contact Frontline Systems, Inc., P.O. Box 4288, Incline Village, NV 89450.
Tel (775) 831-0300 Email info@solver.com Web https://rason.com

Table of Contents

Solver Result Messages 236

RASON DMN/FEEL at Conformance Level 3 248

Appendix 253

Rason Model Components for Optimization or Simulation

Introduction

This section introduces each of the nine different components or sections which make up a RASON model: "variables", "uncertain variables", "data", "dataSources", "engineSettings", "formulas", "modelSettings", "objective", "constraints", "indexSets" and "objective". This chapter explains how each component of your model should be defined. Some optimization models will consist of just 3 sections: `variables` (where the decision variables will be defined), `constraints` (where the constraints will be defined) and `objective` (where the objective will be defined)) where other larger and more complex models might contain several additional segments such as: `engineSettings` (where the engine is chosen and an engine options are specified), `data` (where any arrays used in the calculation of the constraints or objective are defined), `formula` (where any intermediate calculations are performed) and/or `dataSources` (where any data is imported from an outside source such as a CSV file). Most simulation models will be comprised of two components: `uncertainVariables` (where the uncertain variables are defined) and `uncertainFunctions` (where the uncertain functions are defined). However, a simulation model could also contain additional segments such as: `engineSettings`, `data`, `formula`, and/or `datasources`.

Note: The RASON modeling language supports all but a few of Excel's functions[1] which means that you can write a formula easily using functions such as SUM, SUMPRODUCT, etc. along with operators such as + and *. You can define arrays and use Excel functions that return vector and matrix results and access your data from within an Excel worksheet or a database.

Box Functions

Custom defined functions are supported in RASON Decision Services. These custom functions are defined within the boxFunction section of the RASON model but can be reused within any section.

Note: For a list of supported FEEL expressions, see the Decision Tables section that appears later in the guide.

Example:

```
boxFunctions: {
  funPMT: {
    inputs: ['p', 'r', 'n'],
    inputTypes: ['number', 'number', 'number']"},
    language:  "FEEL",
    resultType: "number",
    body: {
      payment: {formula: "(p * r/12)/(1-(1+r/12)**-n)"},
```

[1] Note: Excel functions not supported by the Rason modeling language are: Call(), Cell(), CubeX(), EuroConvert(), GetPivotData(), HyperLink(), Indirect(), Info(), Offset(), RegisterID(), PivotDim(), PivotCube(), FormulaText(), Dollar(), Fixed(), Replace(), Search(), Text() and SqlRequest().
.

```
      fee: {formula: "0.01 * payment"}
  }
  result:  {formula: "payment + fee"}
}
```

Notice the use of the prefix "fun" in the name of the box function, `funPMT`. It is good practice to use this prefix when defining a box function in RASON in order to prohibit a naming conflict error. For example, if this function were instead named "PMT", rather than "funPMT", each time this function was used in the RASON model, the Excel Financial PMT function would be called rather than this defined function.

- inputs (required): Defines the input parameters
- inputTypes (Required if Formula Language = FEEL, otherwise optional): Defines the type for all input parameters.
    - Supported types are:

      Formula Language = Excel

      Array: Any Excel cell reference, i.e. A1:C1.

      Note: This can be used for a Box function that, say, computes the SUMPRODUCT(A1:A3, B1:B3) where A1:A3 is a range for the first input parameter and B1:B3 is a range for the second input parameter.

      Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

      Empty: Select "empty" if Formula Language = EXCEL and no Data Type is being specified.

      Error: Any Excel error such as #N/A, #Number, etc.

      Number: May be an integer or fraction.

      String or Text: Any string

      Language = FEEL

      Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

      Date: Any valid date, such as 05-05-1964

      Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:

      - P for "period"
      - 1D for 1 day
      - T for "time"
      - 1H for 1 hour
      - 2M for 2 minutes and
      - 3S for 3 seconds.
- language (required): Defines the language: FEEL or EXCEL.
- resultType (Required if Formula Language = FEEL, otherwise optional): Defines the type of result returned by the function. The returned value type is specified below the formula language. Custom functions return only one output.
    - Supported types are:

      Formula Language = Excel

      Array: Any Excel cell reference, i.e. A1:C1.

Note: This can be used for a Box function that, say, computes the SUMPRODUCT(A1:A3, B1:B3) where A1:A3 is a range for the first input parameter and B1:B3 is a range for the second input parameter.

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Empty: Select "empty" if Formula Language = EXCEL and no Data Type is being specified.

Error: Any Excel error such as #N/A, #Number, etc.

Number: May be an integer or fraction.

String or Text: Any string

Language = FEEL

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Date: Any valid date, such as 05-05-1964

Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:

- P for "period"
- 1D for 1 day
- T for "time"
- 1H for 1 hour
- 2M for 2 minutes and
- 3S for 3 seconds.

- body (optional): Defines the intermediary formulas used to to calculate the result. In this example, "payment" is calculated as (p * r/12)/(1-(1+r/12)**-n) and "fee" is calculated as 0.01 * payment. Note that "payment" and "fee" are user defined names.
- result (required): Defines the calculation returned by the function. In this example the result adds the payment and fee as defined within the body.

Invocation of Custom Functions

Custom functions are invoked in RASON models as a standard function by name and values passed for parameters in brackets. For example:

```
MonthlyPayment: {formula: "funPMT(amount, rate, term)" finalValue:[]}
```

Notes:

- Variables amount, rate and term are global in scope.
- Custom functions in RASON are polymorphic and multithreaded.

For more information on using custom functions in RASON, see the Using Custom Functions chapter within the RASON User Guide and example models using custom functions under Rason Examples – Decision – Custom Functions.

FEEL Expressions

Variables and constants can be combined through operations called literal expressions. Literal expressions in S-FEEL are similar to formulas in Excel and in the RASON modeling language. For a list of supported FEEL expressions, see the Decision Tables section that appears later in the guide.

LAMBDA Function

The LAMBDA function, introduced in Microsoft Excel, can also be used to create custom, reusable functions which can be invoked using a custom name. For example, a user could use the LAMBDA function to define a new function that calculates a common formula within a RASON model. Using the new function in the cell to calculate the formula, rather than the actual formula, reduces the chance of introducing an error into the model.

Example

```
boxFunctions: {
    "MyLmd": {result: "LAMBDA(y,LET(x,y+1,LET(z,6, z+x*y)))"}
}
```

Notes:

- The number of supported parameters is 253.
- Periods (.) are not supported in LAMBDA function names and parameters.

Notes on the LAMBDA and LET Functions

- **Scope of variables within a LAMBDA function**

  When a LAMBDA expression is calculated, RASON Decision Services will first search the local scope of the expression for the definition of a variable. If not found, RASON will proceed to search the parent scope and then will move to the global scope (cell/range names).

  Assume a variable with the defined name "z" and the custom Lambda function below exist within the same RASON model.

  =LAMBDA (x, y, LET (z, x+1, y * z + b1))

  In this instance, the local scope is the LET function, the parent scope is the LAMBDA function enclosing the LET function, and the global scope is all cell/range names in the RASON model. The variable z =x+1 despite the existence of z in the global scope. The variable z inside LET and the z variable are considered to be different variables.

- **Nesting LAMBDA/LET functions**

  RASON Decision Services supports only nested LET functions. Nexted LAMBDA functions are not supported.

- **Properties of LAMBDA/LET in RASON Decision Services**

  The following list contains important properties of both the LAMBDA and LET functions.

  1. Reusability – LET definitions are not reusable while LAMBDA definitions, even ones containing LET functions, are resuable.
  2. Nesting – LET definitions can be nested while LAMBDA definitions cannot.
  3. Recursion – Although the LAMBDA function is recursive in Excel, RASON Decision Services does not support recursion with this function.
  4. Threads – Both LET and LAMBDA functions running in multiple threads.
  5. Polymorphic Evaluation – The calculation of derivates, intervals, etc inside of a LAMBDA and LET function.
  6. Model Conversion: Both LET and LAMBDA functions are supported in model conversion from Excel to RASON (Create App – RASON).

     - A LET function within an Excel model will appear within the formulas property within the RASON model, after conversion is complete.

       ```
       uncertainFunctions: {
       ```

```
  "c5": { formula: "LET(x, 1+1, LET (y, 2, A5 + x = y))",
mean: []
}
```

- The LAMBDA function within an Excel model will appear within a new section, "boxFunctions" within the RASON model, after the model conversion is complete.

```
boxFunctions: {
  "MyLmd" : {
       result: "LAMBDA(y, LET(x, y+1, LET(z, A13, z+x*7)))"
}
```

For more examples see the RASON User Guide's Using Custom Functions chapter. For example models using the Lambda function, see RASON Examples – Decisions – Custom Functions.

Constraints

This optional section is used for defining normal, recourse or chance constraints in optimization, stochastic optimization or simulation optimization models. There are 10 constraint *input* properties: comment, name, dimensions, type, formula, lowerBound, upperBound, equal, chanceType, and chanceProbability. In return you may ask for the constraint's final value, dual value, dual upper value, dual lower value, slack value intial value and index value. In the example code below, five constraints are defined by a matrix multiplication of the `parts` and `products` arrays. The upper bound of each constraint is contained in the `upper` array. In return, the dual value and upper and lower bounds for the dual value for each constraint will appear in the Result (`dualValue: [], dualUpper:[], dualLower:[]`).

```
constraints: {
cons: {
   formula: "mmult(parts, transpose(products))", upper: [450, 250, 800,
   450, 600], dualValue: [], dualUpper: [], dualLower: [] }
},
```

We also could have created the `cons` block of constraints by using an alternate syntax, shown below. However, if a parameter is defined in this way, it would not be possible to pass (say) new right hand side values outside of the RASON model environment (via a direct call to the RASON REST API).

```
constraints :
    { name: "cons",
      formula: "mmult(parts, transpose(products))",
      upper: [450, 250, 800, 450, 600],
      dualValue: [], dualUpper: [], dualLower: [] }
},
```

Please see the table below for all input properties available in `constraints`.

Input Property	Example	Definition
`aliasName`	`aliasName: “num_parts_inventory”`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver’s Deploy Model button, if a block of cells containing constraint left hand sides is assigned a defined name in the Excel Solver model.

comment	comment: "number of parts used must be less than inventory"	Enter a comment here to describe the constraint or block of constraints. (Optional)
name	name: "constraints"	Enter a name for a constraint or block of constraints. (Optional)
dimensions	1. dimensions: [3,1] 2. dimensions: [3] 3. dimensions: [1,3] 4. dimensions: [3,2]	1. Defines a 2 – dimensional horizontal array with 3 rows and 1 column. 2. Defines a 1-dimensional vertical array with 3 elements. 3. Defines a 2 – dimensional vertical array with 3 elements. 4. Defines a 2 – dimensional array with 3 rows and 2 columns. All arrays are 1 – based. (Optional.) If missing, constraint array shape will be implicitly defined by the shape of the *lower*, *upper*, *equal* or *value* properties, however, for readability of the code, the use of the *dimensions* property is recommended.
type	1. type: "cone" 2. type: "rotatedCone"	1. Defines the constraint or constraint block as belonging to a cone. 2. Defines the constraint or constraint block as belonging to a rotated cone. If chancetype and type arrays are missing, the constraint or block of constraints is assumed to be *normal*. See below for information on cone and rotated cone constraints see the *Using Cone Constraints* topic below.
formula	formula: "mmult(parts, transpose(products))",	Calculates the constraint. (Required.)
lower*	lower: 0 lower: [1, 2, 3] lower: "availInvent" where availInvent is an array of constants.	Specifies the lower bound of the constraint or constraint block. If an array is passed and dimensions, upper, equal or value properties are missing, the shape of the constraint array will be determined by the shape of the lower property. However, it is recommended that the dimensions property be used for readability purposes. If missing, the lower bound is defined as "unbounded". (Optional) Note: Only constant values are supported for this property. If a formula is provided to lower: [], the error: "Can not be parsed" will be returned. If the right hand side of

		your constraint must contain a formula, then simply subtract the RHS from the left hand side of the constraint, for example: x1 + x2 = x3 +x4 can be rearranged to: x1 + x2 – (x3 + x4) = 0.

`upper*`	`upper: 0`
	`upper: [1, 2, 3]`
	`upper: "availInvent"` where availInvent is an array of constants.
		Specifies the upper bound of the constraint or constraint block. If an array is passed and `dimensions, lower, equal` or `value` properties are missing, the shape of the constraint array will be determined by the shape of the `upper` property. However, it is recommended that the `dimensions` property be used for readability purposes. If missing, the upper bound is defined as "unbounded". (Optional)
		Note: Only constant values are supported for this property. If a formula is provided to upper: [], the error: "Can not be parsed" will be returned. If the right hand side of your constraint must contain a formula, then simply subtract the RHS from the left hand side of the constraint, for example: x1 + x2 = x3 +x4 can be rearranged to: x1 + x2 – (x3 + x4) = 0.

`equal*`	`equal: 0`
	`equal: [1, 2, 3]`
	`equal: "availInvent"` where availInvent is an array of constants.
		Defines an equality constraint. If an array is passed and `dimensions, upper, lower` or `value` properties are missing, the shape of the constraint array will be determined by the shape of the `equal` property. However, it is recommended that the `dimensions` property be used for readability purposes. If missing, either `upper` or `lower` must exist. (Optional)

`chanceType`	`chanceType: “VaR”`
	`chanceType: “CVaR”`
	`chanceType: “USet”`
		Defines the constraint or constrant block as a chance constraint(s). Constraint or constraint block *must* contain uncertainties. The property `chanceProbability` property must exist if `chanceType` exists.
		Value at Risk (VaR) – Specifies that the `chanceProbability` percentile of the realizations of the constraint left hand side must be less than or equal to the constraint right hand side; realizations beyond the `chanceProbability` percentile may be greater than the right hand side by any amount.
		Conditional Value at Risk – Specifies that the expected value of all the realizations of the constraint right hand side up to the `chanceProbability` percentile must

		be less than or equal to the constraint left hand side. Uncertainty Set – Applicable only to linear constraints where some or all of the coefficients may depend on the uncertainties. Specifies that the constraint right hand side must be satisfied for all variations from the nominal variable values and do not exceed a bound, measured by a norm. For more information on these types of constraints, see the topics below.
`chanceProbability`	`chanceProbability: 0.95`	Defines the percentile for use with VaR, CVaR, and USet constraints. For more information on chance constraints, see the topics below.

*The RASON Server currently ONLY supports constant values (i.e. 3, 8.54, etc.) or an array containing constant values for the *lower, upper* and *equal* properties.

An output property must be specified within the constraint definition as an empty array.

Output Property	**Example**	**Definition**
`dualLower`	`dualLower: []`	Creates an empty array to hold the Allowable Decrease for the constraint or constraint block. See the topic, *Interpreting Reduced Costs* below for more information on this property.
`dualUpper`	`dualUpper: []`	Creates an empty array to hold the Allowable Increase for the constraint or constraint block. See the topic, *Interpreting Reduced Costs* below for more information on each of these properties.
`dualValue`	`dualValue: []`	Creates an empty array to hold the shadow price for the constraint or constraint block. The shadow price for a constraint is nonzero only when the constraint is binding. See the topic, *Interpreting Reduced Costs* below for more information on each of these properties.
`slackValue`	`slackValue: []`	Creates an empty array to hold the slack value for each constraint. The slackValue holds the constraint's slack which is nonzero only when the constraint is NOT equal to its bound. For example, take the constraint x1 + x2 = 3. If x1 = 0 and x2 = 2, `slackValue` = 1.
`finalValue`	`finalValue: []`	Creates an empty array to hold the final constraint value for the constraint or constraint block.
`initialValue`	`initialValue: []`	Creates an empty array to hold the initial value of the constraint.
`indexValue`	`indexValue: []`	Creates an empty array to hold the index value for each constraint in the block of constraints.

Using Cone Constraints

A simple kind of cone constraint is a non-negativity constraint on a variable or block of variables. These types of constraints specify that the variables must lie within a simple kind of cone, called the *non-negative orthant*. This first order cone places a bound on the *L1*-norm of the vector of decision variables. A second order cone (also called a Lorentz cone or "ice cream cone") is a ***convex*** set that looks like this:

This cone places a bound on the *L2*-norm of the vector of decision variables. If x1, x2, and x3 are variables that lie within this cone, then x1 >= SQRT(SUMSQ(x2, x3)) must hold. A problem with a linear objective andlinear or second order cone (SOC) constraints is called a *second order cone programming* (SOCP) problem; it is always a ***convex*** optimization problem. Second order cone programming is the *natural generalization* of linear programming. It offers the same advantages of convexity and scalability to large problems offered by linear programming – but for a broader class of models. For history buffs, Premium Solver Platform V6.0 was the first commercial software product to offer broad support for second order cone programming.

Please see the two examples below (RGFirehouseLocation.json and RGFirehouseLocationConic.json) which illustrate how to setup the same model in two different ways: the first (RGFirehouseLocation.json) without cone constraints and the second (RGFirehouseLocationConic) with cone constraints. The goal of both models is to find a location, given by x and y coordinates, of a proposed firehouse that minimizes the maximum distance between the firehouse and six cities in the region.

In the first example, there are three decision variables x, y, and z. The x and y variables will hold the final x and y coordinates of the proposed firehouse location. The third variable, z, will be minimized in the objective function.

```
{
    comment: "NLP Example, individual constraints",
    engineSettings : { engine : "GRG Nonlinear" },
    variables : {
       x: { value: 1.0, finalValue: [] },
       y: { value: 1.0, finalValue: [] },
       z: { value: 1.0, finalValue: [] }
    },
    constraints : {
       c1: { formula: "sqrt((x - 1)^2 + (y - 4)^2) - z", upper: 0 },
       c2: { formula: "sqrt((x - 0.5)^2 + (y - 3)^2) - z", upper: 0 },
       c3: { formula: "sqrt((x - 2)^2 + (y - 4)^2) - z", upper: 0 },
       c4: { formula: "sqrt((x - 2)^2 + (y - 2)^2) - z", upper: 0 },
       c5: { formula: "sqrt((x - 2)^2 + (y - 5)^2) - z", upper: 0 },
       c6: { formula: "sqrt((x - 0.5)^2 + (y - 6)^2) - z", upper: 0 }
    },
    objective : {
       obj: { formula: "z", type: "minimize", finalValue: [] }
    }
}
```

The constraints calculate the distance between the proposed firehouse location and each of the six cities using the the Pythagorean Theorem (SQRT ((Xc – X)^2 + (Yc – Y) ^2)), which is a nonlinear function of the variables. The objective function minimizes the z variable to find the smallest possible distance between the firehouse and each city. This model may be solved with the Nonlinear GRG engine.

In the next example, 6 conic constraints (one for each city) is used to calculate the distance between the proposed firehouse location and each city, rather than the Pythagorean Theorem.

```
{
    engineSettings : { engine: "SOCP Barrier" },
    variables : {
         x: { value: 1.0, finalValue: [] },
         y: { value: 1.0, finalValue: [] },
         z: { value: 1.0, finalValue: [] },
         f: { dimensions: [6], value: 1.0 },
         g: { dimensions: [6], value: 1.0 },
         h: { dimensions: [6], value: 1.0 }
    },
    data: {
         corx: { dimensions: [6], value: [1, 0.5, 2, 2, 2, 0.5] },
         cory: { dimensions: [6], value: [4, 3, 4, 2, 5, 6] }
    },
    constraints : {
         dx: { dimensions: [6], formula: "corx - x - g", equal: 0 },
         dy: { dimensions: [6], formula: "cory - y - h", equal: 0 },
         dz: { dimensions: [6], formula: "f - z", upper: 0 },
         cone1: { value: "f[1], g[1], h[1]", type : "cone" },
         cone2: { value: "f[2], g[2], h[2]", type : "cone" },
         cone3: { value: "f[3], g[3], h[3]", type : "cone" },
         cone4: { value: "f[4], g[4], h[4]", type : "cone" },
         cone5: { value: "f[5], g[5], h[5]", type : "cone" },
         cone6: { value: "f[6], g[6], h[6]", type : "cone" }
    },
    objective : {
         obj: { formula: "z", type: "minimize", finalValue: [] }
    }
}
```

The x and y coordinates of each city to be served by the firehouse are given in the `corx` and `cory` arrays, respectively. The `x` and `y` variables will hold the final x and y coordinates of the proposed firehouse. The `f` block of variables will be forced by the `dx` block of constraints to equal the difference between the x coordinate of the proposed firehouse and the x coordinate of each city. The `g` block of variables will be forced by the `dy` block of constraints to equal the distance between the y coordinate of the proposed firehouse and the y coordinate of each city. The `h` block of variables will be driven by the dz block of constraints to be less than or equal to `z`, the variable to be minimized in the objective function. Each member of the `f`, `g`, and `h` variable blocks must (for example `f[1]`, `g[1]`, and `h[1]`) belong to a second order cone constraint which can be rewritten as `f[1] > = SQRT (SUMSQ(g[1]; h[1]`). Minimizing the z variable in the objective function will push the maximum distance between the proposed firehouse location and each city to the lowest possible value guaranteeing that the firehouse is as close as possible to each of the six cities. This model may be solved with the SOCP Engine.

Note: In RASON, you can solve <u>any type</u> of model containing conic constraints. There is no need to select a specific engine within `engineSettings`. If no engine is specified, the model is considered to be nonlinear and an appropriate nonlinear engine will be selected to solve the model. If the model type of SOCP is known, and an engine supporting conic constraints is specified, the model will be solved as an SOCP. Currently, the three engines that support conic constraints are: "SOCP Barrier" (as shown in the above example), "Gurobi Solver" and "Mosek Solver".

Interpreting Reduced Costs

The Shadow Price for a constraint is nonzero only when the constraint is equal to its bound. This is called a *binding* constraint, and its value was driven to the bound during the optimization process. Moving the constraint left hand side's value away from the bound will *worsen* the objective function's value; conversely, "loosening" the bound will *improve* the objective. The Shadow Price measures the increase in the objective function's value *per unit increase* in the constraint's bound. In the case of linear problems, the Shadow Price remains constant over the range of Allowable Increases and Decreases in the variables' objective coefficients and the constraints' right hand sides, respectively. For each constraint, `dualLower` and `dualUpper` will report the constraint right hand will report the amount by which the RHS could be increased or decreased without changing the dual value.

Constraints: Normal, Chance, Recourse

Constraints are relations such as $f(x_1 , x_2 ..., ,x_n)$ <= b, where $x_1 , x_2 ..., ,x_n$ are decision variables. A constraint is *satisfied* when the relation it specifies is true within a small tolerance. When your model includes uncertainty, we must examine how each constraint depends on the uncertainties and the decision variables:

- If a constraint depends only on *certain* parameters and *normal* decision variables, it is 'deterministic' and is handled in the usual way by the Solver. We call this a ***normal* constraint**. (See above for an example.)
- If a constraint depends on *uncertain variable*s and *normal* decision variables, we must specify what it *means* for the constraint to be satisfied. There are many possible *realizations* for the uncertain variables, but only single values for the decision variables. The Solver must find values for the decision variables that cause the constraint to be satisfied for *all*, or perhaps *most* but not all, realizations of the uncertainties. We call this a ***chance* constraint**. For example, we might specify that the constraint must be satisfied 95% or 99% of the time; it can be violated 5% or 1% of the time. For 95%, we denote such a constraint as **VaR $_{0.95}$ A1 <= B1**. But this form may not be your best choice – alternatives called CVaR and USet are discussed in the section "More on Chance Constraints."
- If a constraint depends on *uncertain variable*s and *recourse* decision variables, then the Solver will find an *array* of values for each of the recourse variables, corresponding to the *realizations* of the uncertain vs. Recourse decisions give the Solver flexibility to satisfy constraints that involve uncertainty; but in effect, each such constraint has many *realizations* – one for each realization of the uncertainties. We call this a ***recourse* constraint**.
- A constraint may also depend on *recourse* decision variables, and possibly normal decision variables, but not depend on any uncertain variables. This is also a ***recourse* constraint**, with many *realizations*. The Solver must find values for the recourse variables that satisfy all the constraints where they appear – some with uncertainties, and some without.

Multiple Uncertainties May Offset Each Other

What happens when a constraint depends on *several* different uncertainties? Is such a constraint harder or easier to satisfy than a constraint that depends on just one uncertainty? In the simplest case, suppose we have a *linear* constraint, with coefficients a_i and decision variables x_i: $a_1x_1 + a_2x_2 + ... + a_nx_n \leq b$. Suppose that each coefficient a_i is uncertain (and independent of all the others), with sample values drawn randomly from PsiUniform (a_i – 0.5, a_i + 0.5). The average or *nominal* value of each coefficient is a_i. The 'worst' that can happen is that a sample is drawn where *every* coefficient is a_i + 0.5 – this makes the left hand side (LHS) as large as possible, so it is very likely to violate the condition LHS ≤ b. *But this case is very unlikely to occur*. In *most* realizations of the uncertainties, some coefficients (randomly drawn from the range a_i – 0.5 to a_i + 0.5) will be less than a_i, and some will be greater than a_i. The more uncertainties are involved, the greater the chance that some of them will draw samples less than a_i. If we use a *chance constraint* to specify that the relation must be satisfied (say) 95% or 99% of the time, we actually have a *better* chance of satisfying this constraint when it depends on *many* uncertainties than when it depends on just one – as long as the uncertainties are independent, or at least not highly correlated with each other.

More on Chance Constraints

As explained above, if a constraint depends on *uncertain variable*s and *normal* decision variables, we can seek solution values for the variables that cause the constraints to be satisfied for *all*, or perhaps *most* but not all, realizations of the uncertainties. If we insist that the constraints are satisfied for *all* realizations, we may not be able to *find* values for the decision variables that meet this requirement – and if we do, we will very likely 'pay for this' via worse values for the objective function.

Instead, we can seek solution values for the variables that cause the constraints to be satisfied for *most*, but not necessarily all, realizations of the uncertainties. We might specify that the constraint must be satisfied (it must not exceed a given limit) 95% or 99% of the time; it can be violated 5% or 1% of the time. This is depicted in the chart below, where 95% of the area under the curve is to the left of the bar (i.e. the constraint right hand side value), and 5% is to its right. This is one form of a *chance constraint*; the criterion that it must be satisfied for all realizations of the uncertainties up to a given percentile (say 95%) makes it a *VaR* (Value at Risk) constraint. We write this constraint as **VaR $_{0.95}$ A1 <= B1**.

The RASON modeling languague supports two other criteria besides *VaR* that may be better choices for many models. A chance constraint includes:

- A *left hand side* that depends on decision variables and uncertainties.

- A *relation* that must be either <= or >=. (A *chance* constraint can't be an equality. Note however that a r*ecourse* constraint *can* be an equality.)

- A *type* that may be VaR (Value at Risk), CVaR (Conditional Value at Risk), or USet (Uncertainty Set). These criteria are discussed below.

- A *measure* that may be a percentile 0.01 – 0.99 for VaR or CVaR, or a 'budget of uncertainty' (any positive value) for USet.

Value at Risk Measure

Chance constraints defined by a percentile or VaR (Value at Risk) measure have been used since the early 1960s. Such constraints offer a good deal of modeling flexibility, and they are easy to understand in terms of the probability that the constraint will be satisfied. Value at Risk is used in the banking and securities industries, and its use is mandated by the international Basel II accords. But chance constraints in this form have several drawbacks:

- A VaR constraint with probability 95% requires only that the constraint be satisfied – not violated – 95% of the time; it **says nothing about the magnitude of the violation** that may occur the other 5% of the time. For example, a portfolio of securities that is VaR-constrained to lose no more than $100,000 95% of the time could still lose $1 million+ at other times.

- As a measure of risk, the VaR criterion is **not subadditive**, a property expected of any 'coherent risk measure.' For example, if two portfolios A and B are VaR-constrained to not lose money 95% of the time, it is reasonable to expect that a combined portfolio A+B should have a *95% or better* chance of not losing money – but this is *not* guaranteed by the two portfolio VaR constraints.

- A VaR constraint is **not necessarily convex**; hence, using such a constraint in an otherwise convex model (for example, any linear programming or convex quadratic model) will radically affect its 'solvability' – it means that a globally optimal solution cannot be guaranteed, and solution time may rise exponentially with model size.

Further, when robust optimization methods automatically transform a model with VaR constraints into a larger but deterministic 'robust counterpart' model, it first approximates the non-convex VaR constraint with a convex CVaR constraint, and then transforms the CVaR constraint. Since CVaR is always **more conservative** than VaR as a risk measure, the robust counterpart solution will 'pay a price' in conservativeness, with a worse objective value. Users are often better off using CVaR directly.

Conditional Value at Risk Measure

To deal with the problems of Value at Risk cited above, an alternative risk measure called *Conditional Value at Risk* or CVaR (also called *Expected Tail Loss* or ETL) was developed in the late 1990s. **VaR $_{0.95}$ A1 <= B1** specifies that the 95th percentile of the realizations of A1 must be less than or equal to B1; realizations beyond the 95th percentile may be greater than B1 by any amount. In contrast, **CVaR $_{0.95}$ A1 <= B1** specifies that the *expected value* of ***all*** the realizations of A1 up to the 95th percentile must be less than or equal to B1. Below is a chart that compares VaR and CVaR. VaR is the value (10,000) that lies at the 5th percentile of the realizations of the constraint left hand side; 95% of the realizations are greater than 10,000 and lie in the graph to the right of this point. CVaR (8,000) is the *expected* value (i.e. the mean or average value) of ***all*** the realizations that lie in the 'tail' to the right of the VaR (Note that, if **CVaR $_{0.05}$ A1 <= B1** is satisfied for some B1, then **VaR $_{0.05}$ A1 <= B1** is also (more than) satisfied. As a risk measure, Conditional Value at Risk has several advantages over VaR:

- Unlike VaR, a CVaR constraint at 95% places a **bound on the average magnitude of the violations** that may occur 95% of the time.

- CVaR is a 'coherent risk measure.' It is **subadditive**, so if two portfolios A and B are CVaR-constrained to not lose money 95% of the time, then a combined portfolio A+B has the *same or better* chance of not losing money.

- A CVaR constraint is **always convex**. Models consisting of all convex functions can be solved to global optimality, and solved to very large size using modern interior point optimization methods.

Uncertainty Set Measure

The RASON modeling language supports a third criterion for uncertainty in a chance constraint, which reflects the approach taken in most of the literature on *robust optimization* methods. This criterion, called *USet* (for uncertainty set), is applicable only to *linear* constraints, with coefficients a_i and variables x_i:

$a_1x_1 + a_2x_2 + ... + a_nx_n \leq b$

where some or all of the coefficients a_i may depend on the uncertainties. It is useful to think of the vector [a_1 a_2... a_n] as having a *nominal* or expected value, and a *variation* from this value for each realization of the uncertainties. A constraint of the form **USet$_\Omega$ A1 <= B1**, where $a_1x_1 + a_2x_2 + ... + a_nx_n$ is in A1, and *b* is in B1, specifies that A1 <= B1 must be satisfied for ***all*** *variations* from the *nominal* value of [a_1 a_2... a_n] that do not exceed a bound Ω, measured by a norm. The bound Ω is often called the *budget of uncertainty* for the constraint. A very large Ω says that the constraint must be satisfied for practically all variations of the coefficients from nominal; a Ω of 0 effectively ignores uncertainty, requiring only that A1 <= B1 for the nominal value of [a_1 a_2... a_n], and saying nothing about departures from this value. The RASON modeling language allows you to choose among four different norms to measure variation from the nominal value: The L1, L2, L-Infinity and D norms – as described in "Uncertainty Sets and Norms" that appears earlier in this guide.

Contexts

Use the "contexts" section of a RASON model to define a context: a single object that determines both the type structure of the object and also the value of the object. Components of a context must include two arguments: typeRef which assigns the type and formula to calculate the value. Note that constant values are allowed.

Relation to Custom Types

Context objects resemble component custom types, which are defined below. Component custom types are defined as custom types with components such as names names, types and, optionally, allowed values (i.e. domain). If a component type is attached to a variable, then that variable defines values in its array structure according to the component type description. The variable may be referenced component-wise using the '.' operator. For example,

```
typeDefs: {
    tPmt: {
      components: {
            payment:  { typeRef: 'number' },
            fee:  { typeRef: 'number' },
            total:  { typeRef: 'number' }
      }
    }
}
data: {
      loan: { type: 'tPmt', value: [600000, 1000, 601000], binding: 'get'
}
}
```

Now **loan**.payment, loan.fee, and/or loan.total may be referenced in subsequet formulas within the RASON model. The restriction here is that the variable must define only constant values. These constant values may be obtained using the binding: ‘get’ mechanism or by fetching a record from an external table or even simply including them inline, as shown in the example code above. Regardless of how they are obtained, these values could not be computed. That is, until now. In the latest version of RASON Decision Services, the concept of variables with components has been extended to the next level, context: where components may include formulas.

The Context Definition

A context is a single object which determines the type of the structure and, at the same time, defines the values. Recall that with component types we have two objects – the type and the variable to which we attach the type. This results in an abstract type definition that many different variables may be attached to. However, *context* is a single object encompassing both the type and the variable.

Contexts are defined within the special section **contexts**: { }. Each object has a unique name in the global scope. The context object, "language", defines the formula language using the syntax "language": "Excel" or "language": "FEEL". (Currently, only FEEL and Excel are supported formula types.)

Components are defined by a unique name in the local scope and through component properties.

The **typeRef/type** property must be a supported type in Excel or FEEL; custom types are not allowed. Each component must have either a **value** or **formula** property to define the value attached to it. These two properties plus the holding mechanism, makes the context type variables distinguishable from the component type variables. See the example below.

```
contexts: {
    cPmt: {
        language: "FEEL",
        components: {
          payment: {
            typeRef: 'number',
            formula: "(loan.principal*loan.rate/12)
            / (1 - (1 + loan.rate/12)**-loan.termMonths)"
          },
          fee: {
            typeRef: 'number',
            value: 10
          },
          total: {
            typeRef: 'number',
            formula: "payment + fee"
          }
        }
    }
}
```

The components “payment”, “fee”, and “total” are in the local scope of the context variable cPmt. Outside that context, the same names can be used either in the global scope or another local scope and they will be distinguishable. Notice that the formulas in the context can reference variables from both local and global scopes.

Defined in this way, the context object is a variable itself and can be used in formulas as a whole – it will be treated as a vertical array with all component values. The context object may also be used component-wise through the ‘.’ operator. For example, **cPmt**.total will return only the **total** component.

The component property **typeRef** comes from the DMN syntax, but the Excel property **type** is also supported. Since the basic component types depend on the language, users must pay attention to the assigned type values. For example, if a context uses language: Excel, then the FEEL type "duration" may not be assigned to a component. Frontline encourages the usage of *typeRef* with language: "FEEL" and *type* with language: "Excel".

Notice that the context object **resembles box functions** without arguments. The difference between the two is that box functions have one more result formula and that is the only value they are able to compute in return. Box functions reference box function components as results. Since contexts have no result or default components, when used without the ‘.’ operator, contexts return all component values in a vertical array.

There are two functions which can be used optionally with Context objects when the language is set to FEEL, i.e "language": "FEEL". RASON Decision Services has implemented them in order to obtain compliance for the DMN specification.

- **getValue**(contextObject, componentName) is equivalent to **contextObject**.componentName
- **getEntries**(contextObject) is equivalent to simply referencing the **contextObject**

Custom Types

In past versions of RASON, type definitions were not required as optimization and simulation models dealt exclusively with numeric values. However, with the recent introduction of decision tables and custom functions, RASON Decision Services is now supporting custom type definitions. With this new service, RASON Decision Services now conforms to **DMN Specification Level 2**. Custom Type definitions can be applied to all sorts of RASON model problem types including optimization, simulation and stochastic models along with decision tables and custom functions.   Note: Custom Type definitions are not supported in RASON data mining models.

For more information on Custom Types or to read through a few example models using custom types, please see the Custom Types Definitions chapter within the RASON User Guide.

Data-source Binding

One key benefit to using a custom type definition is the ability to bind a single variable-structure, containing multiple named components, to the entire record.

In the past, the following data source declaration would require three different variables in order to bind to each value column.

```
"datasources": {
  "dsc_loan": {
    "type": "csv",
    "connection": "loan_data.txt",
    "selection": "loanID=?",
    "parameters":  {
      "ID": {
        "binding": "get",
        "value": "L1"
      }
    },
    "indexCols": ["loanID"],
    "valueCols": ["principal", "rate", "termMonths"]
  }
}
```

However, once a custom type definition has been defined…

```
"typeDefs": {
"tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [">0"]},
      "rate": {"typeRef": "number", "allowedValues": ["0..1"]},
      "termMonths": {"typeRef": "number", "allowedValues": ["0>"]}
 }
  }
}
```

…then a new variable can be introduced with "type" set to the custom type definition (in this example "loan") and that variable can be bound to the data-source (in this example "dsc_loan) as shown in the code below.

```
"data": {
  "loan": {
    "type": "tLoan",
    "binding": "dsc_loan"
   }
}
```

The binding property feeds the components of the variable, loan, with the values in the data-source record. Later in formulas, the components may be referenced through the "." operator, for example:

```
"formulas": {
  "payment": "(loan.principal * loan.rate/12)/(1-(1+loan.rate/12)^-
   loan.termMonths)",
  "finalValue":[]

}
```

Note: The variable "loan" of this custom type definition can be alternatively initialized inline or through the existing binding "get".

```
"data"
  "loan": {
    "type":  "tLoan",
    "value": [100000, 0.0375, 360],
    "binding": "get"
  }
}
```

Custom Types in RASON

Custom Type is a major feature in the DMN/FEEL specification (Conformance Level 2) utilized heavily in the development of Decision Trees and Custom Functions. However, this feature may be used in RASON Decision services beyond these two applications.

There are two different structures for custom types: custom types with constraints on values and custom types with components.

- Custom types with constraints on values

In this structure, all members of tEmploymentStatus and tAge are of the same typeRef, either "string" for tEmployementStatus or "number" for tAge.

Custom Type with Constraints Example

```
"typeDefs": {
  "tEmploymentStatus": {
    "type": "string",
    "allowedValues": ["UNEMPLOYED", "EMPLOYED", "SELF-EMPLOYED",
    "STUDENT"]
  },
  "tAge": {
    "language": "FEEL",
    "typeRef": "number",
    "allowedValues": ["[18..21]", ">65"]
  }

},
```

- Custom types with components

  This custom type uses the components property to define a list of components for the custom type structure. Notice that this structure allows different types to be passed to each component in the type definition.

Custom Type with Components Example

```
"typeDefs": {
  "tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [">0"]},
      "rate": {"typeRef": "number", "allowedValues": ["0..1"]},
      "termMonths": {"typeRef": "number", "allowedValues": ["0>"]}
  }
}
```

Custom Type Specifications

A custom type must be defined within the `"typeDefs": {}` section of the RASON model.

The components of a custom type definition are:

- `"language"`: Select the syntax (Excel or FEEL) by using `"language": "FEEL"` or `"language": "Excel"`. The supported type is determined by the language setting. If missing, the default is "Excel". Type definitions within the same RASON model can be different. In other words, two type definitions within the same RASON model using two different language settings may exist.
- `"isCollection"`: Use the "isCollection" property to allow multiple records to be passed to the variable with the given "type". See the Advanced Features section below for more information on this property.
- `"typeRef"` or `"type"`: Assigns a variable to a given type.
- `"language":  FEEL` or `Excel`
- If `"language": "FEEL"`, use the `"typeRef"` property.

```
"typeDefs": {
  "tEmploymentStatus": {
    "language": "FEEL",
    "typeRef": "string",
    "allowedValues": ["UNEMPLOYED", "EMPLOYED", "SELF-EMPLOYED",
    "STUDENT"]
  }
}
```

- If "language": "Excel", use the "type" property rather than "typeRef".

```
"typeDefs": {
  "tEmploymentStatus": {
    "language": "Excel",
    "type": "string",
    "allowedValues": ["UNEMPLOYED", "EMPLOYED","SELF-EMPLOYED",
     "STUDENT"]
  }
   }
```

<u>Supported Types when Formula Language = Excel</u>

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Number: May be an integer or fraction.

String or Text: Any string

<u>Support Types when Language = FEEL</u>

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Date: Any valid date, such as 05-05-1964

Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:

- P for "period"
- 1D for 1 day
- T for "time"
- 1H for 1 hour
- 2M for 2 minutes and 3S for 3 seconds.

  Note: Since the basic component types depend on the "language" (Excel or FEEL) used, it is important for users to note the assigned type values. For example, if "language": "Excel", "typeRef" may not be set to "duration" since this type reference is not supported for this language.

- "components": Use this property to list the members in the type definition. This example contains 3 components: principal, rate, and termMonths. Each of these components is of type "number".

```
"typeDefs": {
  "tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [">0"]},
      "rate": {"typeRef": "number", "allowedValues": ["0..1"]},
      "termMonths": {"typeRef": "number", "allowedValues": ["0>"]}
    }
  }
},
```

- "allowedValues": Use this property to specify the exact values that a type definition can take on, for example, a value greater than 0. In this example,

```
"typeDefs": {
  "tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [">0"]},
      "rate": {"typeRef": "number", "allowedValues": ["0..1"]},
      "termMonths": {"typeRef": "number", "allowedValues": ["0>"]}
    }
  },
```

Data

Data arrays may be defined and calculated in this optional section to be used later when defining constraints, the objective in an optimization model, or an uncertain function in a simulation model. If you are pulling data from an external source, use this section to "bind" the data to an array or table.

In the example code below, data from the `qty` column from the `parts_data` data source is assigned to the `parts` table. Note: A table is created here, rather than an array, by the use of the `valueCol` property.

```
data: {
        parts: {
            binding: "parts_data", valueCol: 'qty'
        },
}
```

Scalars, arrays or tables containing scalars maybe be defined in the data section to be used in a constraint, objective or uncertain function definition.

The following is an example of a scalar constant, which is neither an array nor a table.

```
time: { value: 10 }
```

In the example below, the array `profit` with size equal to 3 contains the values, 75, 50, and 35. In this instance, the `binding` property allows write access to the `profit` array outside of the model environment.

```
data: {
profit: {
            dimensions: [3], value: [75, 50, 35], binding: "get" },
},
```

The following is an example of a scalar constant, which is neither an array nor a table.

```
time: { value: 10 }
```

To change the array elements in profit to 100, 75, 50; you can pass new data directly in the REST API call, via standard HTTP GET parameters, for example:

```
$.get(https://rason.net/api/optimize?profit=100,75,50...
```

To change only one element, say the middle element from 50 to 60, your call to the REST API, via standard HTTP GET parameters would change to:

```
$.get(https://rason.net/api/optimize?profit[2]=...
```

We also could have created the profit array by using an alternate syntax, shown below. However, when a parameter is defined in this way, you will not be able to pass new values to the array outside of the RASON model environment (as shown above).

```
"data" : [
  { name: "profit", value: [75, 50, 35], binding: "get", finalValue: [] }
 ],
```

All properties available for `data`, can be found in the table below.

Data Property	Type	Explanation
`aliasName`	`aliasName: “num_parts_inventory”`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver’s Deploy Model button, if a block of cells containing data is assigned a defined name in the Excel Solver model.

`binding`	`binding: "get"`
	`profit: { binding: "profit_data" }`
		Allows data to be edited outside of the model from a URL or when calling the RASON^{TM} interpreter to solve an optimization or simulation model.
		Used to bind imported table from the `profit_data` datasource to a new table named `profit`.

`comment`	comment: "partsReq" array holds the number of parts required to produce each product	Enter a comment here to describe the data.
`name`	`name: "parts"`	Use this property to define the table, array or scalar name.
`type`	`p: { type: ‘number’, value: 1, binding: ‘get’ }`	Use the `type` property to ensure that the correct data type is passed with the binding ‘get’. Valid standard types are: "boolean", "number", "string", "array", "array/boolean", "array/number", "array/string" “array” - array of any data “array/number” - array of numbers only or a scalar number Data Mining types: "dataset", "fittedModel"
`value`	value: [1, 1, 1] value: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]	Sets the values of the array. Sets the values of a table. If dimensions property is missing, the shape of the variable array will be determined by the shape of the value property. However, it is recommended that the *dimensions* property be used for readability purposes.
`valuecol`	valueCol: ['initials']	Used with `binding` property to bind imported values from a readable data source. If omitted, the RASON interpreter assumes the last column in the table as the input to `valueCol`.

Data Sources

External data sources may be defined in this optional section. Data from these sources is imported into parametric tables or arrays to be used in 1. formula calculations or 2. as initial starting points for decision variables in a nonlinear optimization model. Currently the RASON modeling language supports ten different data sources: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "odata" (OData database), "mssql" (Microsoft Sequel), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file). Data sources such as "Access", "ODBC", "CSV", etc, contain data in tables with records described by index and value columns. Binding to these data sources results in table objects. Data source types such as Excel and CSV may contain data in 2-dimensional arrays without any descriptions. Binding to these data sources results in array objects. Objects are bound to data sources within the `data` section. However, if exporting the results of a solve, we must bind to objects within the variables, constraints, objective, uncertainVariables, and uncertainFunctions sections.

Importing

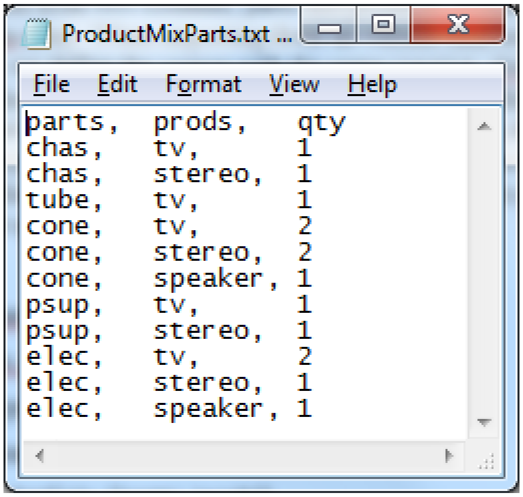
In the example below, data from three columns, "parts", "products" and "qty", within the ProductMixParts.txt CSV file is imported to the data source `parts_data`. (To open ProductMixParts.txt, browse to (typically) C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON.)

The first property, `type`, specifies the type of file where the data is contained. In this example, the file is a CSV (Comma Separated Values) file as shown in the screenshot. The second property, `connection`, specifies the file name within quotes (`"ProductMixParts.txt; header"`). The term `header` appears after the file name because the CSV file contains column headings. If your CSV file does not contain column headings, this term should be omitted. (See Note below.) The term `direction` stipulates whether the contents of the file are being imported or exported. If importing, then direction should be `import`, the default setting for "direction".

The 3rd property indexes the data first by the `parts` column and secondly by the `prods` column using `indexCols`. The 4th property defines the value column (column containing values rather than text), `qty`, using `valueCols`. *The property `indexCols` **must** appear before `valueCols`. The order of columns listed by `indexCols` should be the same as the order in the datasource selection.*

Note: The properties indexCols and valueCols describe a RASON Table while colIndex and rowIndex describe a dataframe. These properties should be be mixed.

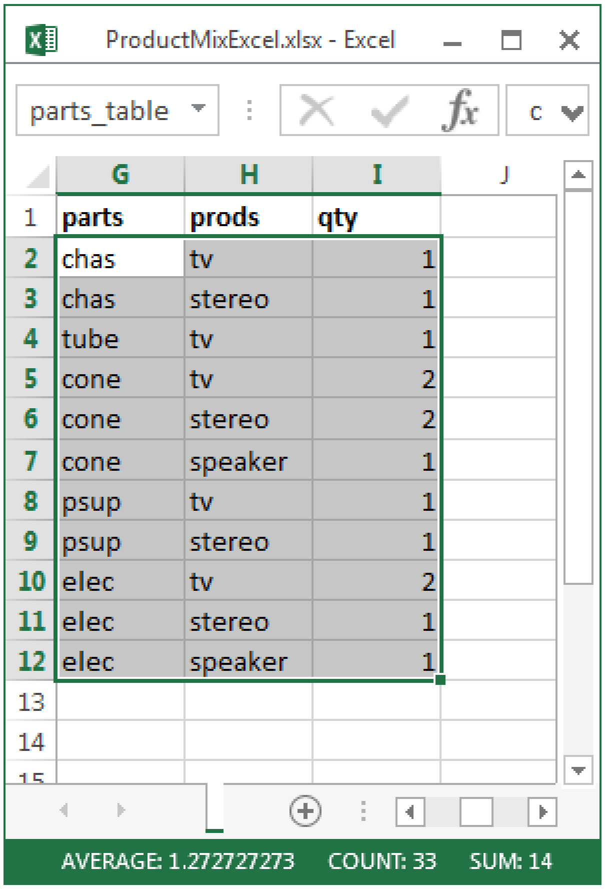
Note: Specifying that your CSV file contains column headings in the `selection` property is specific to CSV files, this is not needed when using an Access or ODBC database or when your data is contained in an Excel file.



```
parts,  prods,    qty
chas,   tv,       1
chas,   stereo,   1
tube,   tv,       1
cone,   tv,       2
cone,   stereo,   2
cone,   speaker,  1
psup,   tv,       1
psup,   stereo,   1
elec,   tv,       2
elec,   stereo,   1
elec,   speaker,  1
```

```
   datasources: {
        parts_data: {
            type: "csv",
            connection: "ProductMixParts.txt; header",
            indexCols: ["parts", "prods"],
            valueCols: ["qty"],
            direction: "import"
        }
},
```

In the screenshot below, we have entered the same data as in the CSV file above into a spreadsheet in Excel.



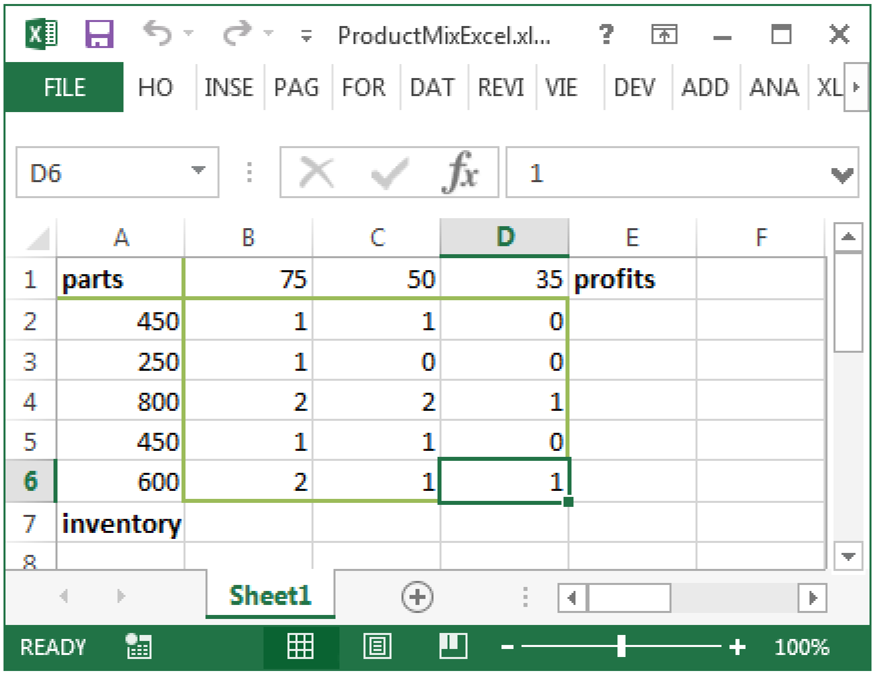
	G	H	I	J
1	parts	prods	qty	
2	chas	tv	1	
3	chas	stereo	1	
4	tube	tv	1	
5	cone	tv	2	
6	cone	stereo	2	
7	cone	speaker	1	
8	psup	tv	1	
9	psup	stereo	1	
10	elec	tv	2	
11	elec	stereo	1	
12	elec	speaker	1	
13				
14				

AVERAGE: 1.272727273 COUNT: 33 SUM: 14

In this example, the first property, `type`, specifies that the data is contained in an Excel file. The second property, `connection`, specifies the name of the file, `"ProductMixExcel.xlsx".` The 3rd property, `selection: "Parts_Table"`, is a defined name given to the Excel range G2:I12. Alternatively, we could also pass `selection: "Sheet1!G2:I12"`. The 4th property, `indexCols`, indexes the data first by the `"parts"` column then by the `prods` column. The 5th property, `valueCols`, defines the value column (column containing values rather than text), `qty.` The 6th property, `direction`, specifies that the contents of the data source are being imported, the default setting.

```
datasources :
    {
      parts_data:  {
            type: "excel",
            connection: "ProductMixExcel.xlsx",
            selection: "parts_table",
            indexCols: ["parts", "prods"],
            valueCols: ["qty"],
            sortIndexCols: true,
            direction: "import"
            }
      }
```

If we were to add more data to `parts_table`, then at a minimum we would need to update the `selection` property. Here's the same data but this time the data is "raw", in other words, all the columns contain values. In this instance we can create an indexed set and define column and row headings using the properties `colIndex` and `rowIndex`. Now, if a new product or part is added, this section of our model will not require any changes.



D6 | 1

	A	B	C	D	E	F
1	parts	75	50	35	profits	
2	450	1	1	0		
3	250	1	0	0		
4	800	2	2	1		
5	450	1	1	0		
6	600	2	1	1		
7	inventory					

As in the example above, the first property, `type`, specifies that the data is contained in an Excel worksheet, `"excel"`; the second property, `connection`, passes the name of the Excel file, `"ProductMixExcel.xlsx"`; and the third property, `selection`, passes the Excel cell range that contains the data, in this instance, `"Sheet1!B2:D6"`.

However in this example, a dataframe is created for the parts_data datasource using the two properties colIndex and rowIndex. When a parameter is binding to such a datasource, the object is a dataframe or a 2D array.

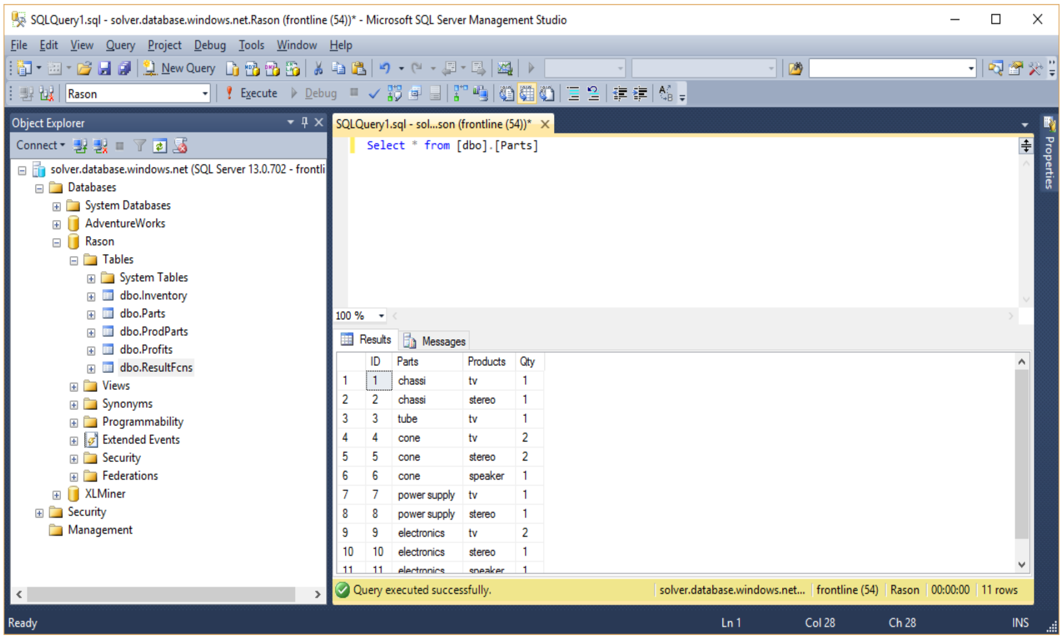
A dataframe, the workhorse of the Rason Server, is a collection of data organized into named columns of equal length and homogeneous type. Rason uses DataFrames to deliver input data to an algorithm and to deliver the results of the algorithm back to the user. DataFrames hold heterogeneous data across columns (variables): numeric, categorical, or textual. When solving a decision flow containing optimization or simulation models, the columns that are indexed over the same dimensions and that belong to the same entity are reported in a single dataframe with multiple columns rather than multiple dataframes, i.e. final, dual, initial, etc for optimization results and statistics for uncertain variables or functions in simulation models. RASON can still bind to the individual results such as optModel.x.finalValue but will also consider the possibility of the last segment being a dataframe column rather than a separate dataframe. As a result, JSON responses are concise which greatly simplifies OData representation and querying.

The 4th property, `colIndex`, binds the index name `prods` to the columns and the 5th property, `rowIndex`, binds the index name `#parts` to the rows. The property `colIndex` binds a set of integers from 1 to the number of columns and the property `rowIndex` binds a set of integers from 1 to the number of rows to the 2-dimensional array `parts_data`.

```
datasources :
    {
      parts_data:  {
            type: "excel",
            connection: "ProductMixExcel.xlsx",
            selection: "Sheet1!B2:D6",
            colIndex: "prods",
            rowIndex: "parts",
            direction: "import"
            },
}
```

In this example, if a new product or new part is added, there will be no changes required to this section of the model. It is completely scalable.

This next example illustrates how to import data from an SQL database residing on an Azure server in the Cloud using an ODBC connection string. (See the example RGProductMixSQL11.json on www.RASON.com.)



```
Select * from [dbo].[Parts]
```

	ID	Parts	Products	Qty
1	1	chassi	tv	1
2	2	chassi	stereo	1
3	3	tube	tv	1
4	4	cone	tv	2
5	5	cone	stereo	2
6	6	cone	speaker	1
7	7	power supply	tv	1
8	8	power supply	stereo	1
9	9	electronics	tv	2
10	10	electronics	stereo	1
11	11	electronics	speaker	1

Query executed successfully. | solver.database.windows.net... | frontline (54) | Rason | 00:00:00 | 11 rows

The RASON modeling language allows readable and writeable access to outside data sources, such as an SQL database residing on a an Azure server in the Cloud using an ODBC connection string. Note that within the datasources section, data is matched by name using the `indexcols` and `valuecols` properties rather than by position, i.e. see `selection` within `parts_data`, in the example code below   (See the example RGProductMixSQL11.json on www.RASON.com.)

The first property, `type`, specifies the type of file containing the data, in this case the file is a SQL database. The second property, `connection`, passes the connection string as obtained from the server. (See below for information on creating a Named Data Connection.) The third property, `selection`, imports three fields from the Parts table, `Parts`, `Products` and `Qty` ordered according to the `ID` field. The 4th property, `indexCols`, indexes the data first by `parts` and secondly by `prods` while the 5th property, `valueCols`, holds the actual data from the `qty` field. The 6th property stipulates the "direction" of the file as "import".

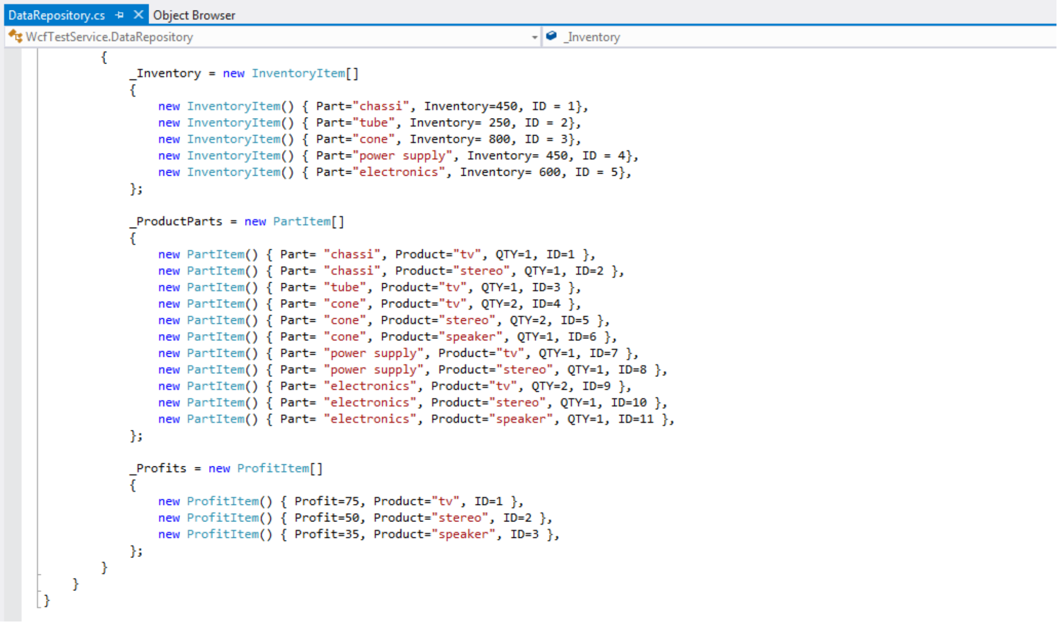
```
datasources :


      parts_data:  {
              type: "odbc",
              connection: "Driver={SQL Server Native Client 
      11.0};Server=tcp:solver.database.windows.net,1433;Database=Rason;Uid
      =rasonread;Pwd=Rason1234;Encrypt=yes;TrustServerCertificate=no;Conne
      ction Timeout=30;",
              selection: "SELECT Parts as parts, Products as prods, Qty as 
      qty FROM Parts ORDER BY ID",
              indexCols: ['parts', 'prods'],
              valueCols: ['qty'],
              direction: "import"

      },
```

We also could have created the `parts_data` data source by using an alternate syntax, shown below. However, when a variable is defined in this way, it will not be available outside of the model environment.

```
datasources : {
    { name: "parts_data",
      connection: "Driver={SQL Server Native Client
      11.0};Server=tcp:solver.database.windows.net,1433;Database=Rason;Uid
      =rasonread;Pwd=Rason1234;Encrypt=yes;TrustServerCertificate=no;Conne
      ction Timeout=30;",
      selection: "SELECT Parts as parts, Products as prods, Qty as qty
      FROM Parts ORDER BY ID",
      indexCols: ["parts", "prods"],
      valueCols: ["qty"],
      direction: "import"
     }
},
```

Our final example illustrates how to import data from an OData data source. This model is also completely scalable. For more information on OData, see <a ref="http://www.odata.org" target="_blank">www.odata.org</a>.   Note: OData data sources are not currently writeable due to limitations in the commonly OData specification.

A screenshot of the OData data source can be found below. (To open this example, browse to (typically) C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON and open the file ProductMixOData1.json.)



```
            {
                _Inventory = new InventoryItem[]
                {
                    new InventoryItem() { Part="chassi", Inventory=450, ID = 1},
                    new InventoryItem() { Part="tube", Inventory= 250, ID = 2},
                    new InventoryItem() { Part="cone", Inventory= 800, ID = 3},
                    new InventoryItem() { Part="power supply", Inventory= 450, ID = 4},
                    new InventoryItem() { Part="electronics", Inventory= 600, ID = 5},
                };

                _ProductParts = new PartItem[]
                {
                    new PartItem() { Part= "chassi", Product="tv", QTY=1, ID=1 },
                    new PartItem() { Part= "chassi", Product="stereo", QTY=1, ID=2 },
                    new PartItem() { Part= "tube", Product="tv", QTY=1, ID=3 },
                    new PartItem() { Part= "cone", Product="tv", QTY=2, ID=4 },
                    new PartItem() { Part= "cone", Product="stereo", QTY=2, ID=5 },
                    new PartItem() { Part= "cone", Product="speaker", QTY=1, ID=6 },
                    new PartItem() { Part= "power supply", Product="tv", QTY=1, ID=7 },
                    new PartItem() { Part= "power supply", Product="stereo", QTY=1, ID=8 },
                    new PartItem() { Part= "electronics", Product="tv", QTY=2, ID=9 },
                    new PartItem() { Part= "electronics", Product="stereo", QTY=1, ID=10 },
                    new PartItem() { Part= "electronics", Product="speaker", QTY=1, ID=11 },
                };

                _Profits = new ProfitItem[]
                {
                    new ProfitItem() { Profit=75, Product="tv", ID=1 },
                    new ProfitItem() { Profit=50, Product="stereo", ID=2 },
                    new ProfitItem() { Profit=35, Product="speaker", ID=3 },
                };
            }
        }
}
```

The `dataSources` section contains the following code:

```
datasources : {
parts_data:  { type: "odata", connection:
"http://localhost:60865/MyWcfDataService.svc/", selection:
"ProductParts?$format=json&columns=Part,Product,QTY",
indexCols: ['parts', 'prods'], valueCols: ['qty'], direction: "import" },
invent_data: { type: "odata", connection:
"http://localhost:60865/MyWcfDataService.svc/", selection:
"Inventory?$format=json&columns=Part,Inventory",
indexCols: ['parts'], valueCols: ['inventory'], direction: "import" },

profit_data: { type: "odata", connection:
"http://localhost:60865/MyWcfDataService.svc/", selection:
"Profits?$format=json&columns=Product,Profit",
indexCols: ['prods'], valueCols: ['profit'], direction: "import" }
},
```

Let's look at the `parts_data` data source first.

The first property for `parts_data`, `type`, specifies the type of file containing the data. In this case, the type is an OData data source. The second property, `connection`, specifies the location of the OData data source on the internet or distributed server. (See below for information on creating a Named Data Connection.) The The third property, `"ProductParts?$format=json&columns=Part,Product,QTY"`,imports three fields from the `ProductParts` table, `Part`, `Inventory` and `QTY`. In this example, `$format=json` is passed within the selection property to stipulate which OData format (JSON or XML) the table should be returned. This is an optional argument. If passed, the OData service will return the data in the format specified, $format=json for JSON or $format=atom for XML. If omitted, the OData service will return the data in preferred format: JSON, XML. The RASON server will automatically recognize the format if not specified. The 4th property, `indexCols`, indexes the data first by `parts` and secondly by `products`, while the 5th property, `valueCols`, imports the actual data from the `qty` field. The `ID` column within the `ProductParts` table is not used. The 6th property, `direction`, indicates that the data will be "imported".

The properties for the `invent_data` and `profit data` data sources are similar. The first property, `type`, specifies the type of file, the second property `connection` specifies the location of the OData data source on the internet or server, while the third property, `selection`, specifies the columns to be imported and in what format they should be imported, in both cases, JSON. In `invent_data`, the `Part` and `Inventory` fields are imported from the `Inventory` table and in `profit_data`, the `Product` and `Profit` fields are imported from the `Profits` table. The fourth property, `indexCols`, indexes the data by `parts` in `invent_data` and `prods` in `profit_data`. The last property, `valueCols`, imports the actual data from within the `inventory` (in the `invent_data` source file) and `profit` (in the `profit_data` source file) fields.

Using a Named Data Connection

In previous versions of RASON, models that accessed external databases required actual credentials to be passed, such as database URLs, port numbers, usernames, and passwords, in the text of the RASON model, in a dataSource declaration, as shown above and in the example code below.

Previous versions of RASON

```
 "parts_data": {
      "type": "odbc",
      "connection": "Driver={SQL Server Native Client
   11.0};Server=tcp:solver.database.windows.net,1433;Database=Rason;Uid=ra
   sonread;Pwd=Rason1234;Encrypt=yes;TrustServerCertificate=no;Connection
   Timeout=30;",
      "selection": "SELECT Parts as parts, Products as prods, Qty as qty
   FROM Parts ORDER BY ID",
      "indexCols": [ "parts", "prods" ],
      "valueCols": [ "qty" ],
      "direction": "import"
       },
```

RASON 2020 offers an alternative to tackle this security risk by substituting

```
"connection": "Driver={SQL Server Native Client
   11.0};Server=tcp:solver.database.windows.net,1433;Database=Rason;Uid=ra
   sonread;Pwd=Rason1234;Encrypt=yes;TrustServerCertificate=no;Connection
   Timeout=30;",
```

with three options: a file containing the contents of "connection" as in (1) below, a named Data Connection as shown in (2) or a URL pointing to Microsoft Common Data Service as shown in (3).

1. `"connection":  "File = filename",`

   RASON 2020 will interpret this as (i) get the text contents of `filename`, which must be attached to the current model instance and (ii) substitute this text for the string `"File=filename"`. Therefore, if `filename` contains the text `"Driver={SQLServerNativeClient…Timeout=30;"`, the effect will be the same as in previous versions of RASON.

2. `"connection": "Name=myname",` where myname is the name given to the Data Connection. See below for instructions on how to create a named Data Connection.

3. `"connection": "secret=uri",` where uri is the Microsoft Common Data Service URL

   `"connection": "xxxx.crm.dynamics.com"` where the actual Microsoft Common Data Service URL is passed directly to "connection".

   If using a with `"secret=url"` in the dataSources section of your RASON model, enter a URI of the form https://subdomain.crm.dynamics.com. , i.e. `"https://www.cdatacloud.net/solver/api.rsc/GoogleSheets1_ODataSourceExample_Sheet1",`

   RASON 2020 will interpret this as (i) get the text contents of the "secret" represented by the URL and (ii) substitute this text for the string "Secret=url". So if the "secret" contains the text "Driver={SQLNativeClient…Timeout=30;", the effect will be the same as in previous versions of RASON.

   Similarly, if using CData Cloud Hub with `"connection": "xxxx.crm.dynamics.com"`, enter a URI of the form https://subdomain.crm.dynamics.com.

   RASON 2020 will interpret this as (i) get the text contents of the connection represented by the URL and (ii) substitute this text for the string "connection".

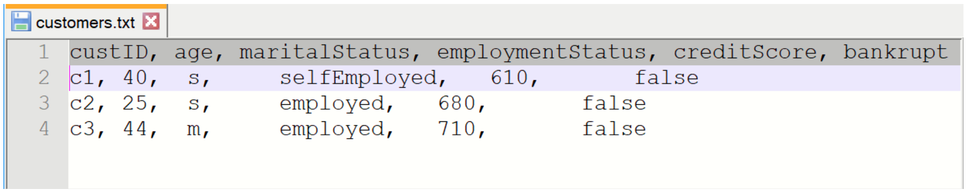
   Currently, RASON 2020 supports "secrets" maintained, only, in an Azure Key Vault. Enterprise customers can provision their own Key Vault and arrange to authenticate the RASON Server to this Key Vault if so desired.

For more information on how to setup and maintain a named Data Connection, see the RASON Services WEB IDE chapter within the RASON User Guide.

Parametric Selection Feature

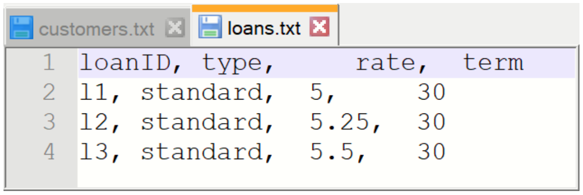
The example model DT Loan Strategy Model2.json, demonstrates how to import this same data from an external data file using a parametric selection criteria. A Parametric Selection allows a single record to be selected from an external datasource file as an input. *Parametric selection in data-sources is universal, but it is critical to decision tables, which expect a single record for their inputs. All supported data types may be used with this feature.*

Much of the customer and loan data is imported from the two datasources: loan_data and cust_data. The datasources section creates two datasources, cust_data and loan_data. The datasource, cust_data, binds to the customers.txt csv file. This file contains five input parameters, age, maritalStatus, employmentStatus, creditScore and bankrupt. A screenshot of this file is shown below.



```
custID, age, maritalStatus, employmentStatus, creditScore, bankrupt
c1, 40,  s,     selfEmployed,   610,       false
c2, 25,  s,     employed,   680,       false
c3, 44,  m,     employed,   710,       false
```

The loan_data datasource binds to the loans.txt csv file. This file contains three input parameters: type, rate, and turn.



```
loanID, type,     rate,  term
l1, standard,  5,     30
l2, standard,  5.25,  30
l3, standard,  5.5,   30
```

```
{ modelName: "loanStrategy",
  datasources :{
      cust_data:  {
         type: "csv",
         connection: "customers.txt",
         selection: "custID = ?",
         parameters: {
            cuID: {
               binding: 'get',
               value: 'c1'
            }
       },
      indexCols: ['cust ID'],
      valueCols: ['age', 'maritalStatus', 'employmentStatus',
      'creditScore', 'bankrupt'],
      direction: "import"
},
```

Inside of the cust_data datasource, we see the connection argument passing the CSV file, `connection: "Customers.txt"` (screenshot above). The `selection` argument selects the "CustID" column , from the Customers.txt file, and replaces "custID = ?" with "custID = cuID"; the parameters argument binds "cuID" to "c1".   In addition, `indexCols` is set to "cust ID" and `valueCols` are set to 'age', 'maritalStatus', 'employmentStatus', 'creditScore', and 'bankrupt'. This means that cust ID is the index column and 'age', 'maritalStatus', 'employmentStatus', 'creditScore', and 'bankrupt' are the value columns.

```
      loan_data:  {
         type: "csv",
         connection: "loans.txt",
         selection: "loanID = ?",
         parameters: {
            loID: {
               binding: 'get',
               value: 'l1' }
         },
         indexCols: ['loanID'],
         valueCols: ['type', 'rate', 'term'],
         direction: "import" }
      },
```

Inside of the loan_data datasource, we see the connection argument passing the CSV file, `connection: "loans.txt"` (screenshot above). The `selection` argument selects the "loanID" column , from the loans.txt file, and replaces "custID = ?" with "custID = loID"; the parameters argument binds "loID" to "l1". In addition, `indexCols` is set to "loanID" and `valueCols` are set to 'type', 'rate' and 'term'. This means that loanID is the index column and 'type', 'rate and 'term' are the value columns.

If we wanted to use multiple selection criteria, such as cust ID and maritalStatus, we would change the code to the following:

```
cust_data:  { type: "csv",
              connection: "customers.txt",
              selection: "custID = ? and maritalStatus=?",
              parameters: {
                 cuID: { binding: 'get', value: 'c1' },
                 marryStat: {binding: 'get', value: 'c2'}
              },
              indexCols: ['custID'],
              valueCols: ['age', 'maritalStatus', 'employmentStatus',
              'creditScore', 'bankrupt'],
              direction: "import"
 },
```

The RASON Server will map "custID=?" with "custID = c1" and "maritalStatus=?" with "maritalStatus = m" using the order found in the `selection` and `parameters` arguments, i.e. `custID` precedes `maritalStatus` in `selection` thus `cuID` must precede `marryStat` in `parameters`. A query can return any number of rows that satisfy the filtering condition, from 0 to infinity. Note that parameter names must not be the same as parameter names in the selection query, i.e. "cuID" could not be renamed to "custID".

To make this same query outside of the RASON model, use:

```
$.get(https://rason.net/api/decision?cuID=c1&marryStat=s.....
```

Or in general:

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....
```

Note: In this instance, quotes are not needed around the value arguments (in this case c1 and s). Quotes would only be needed if a string with spaces were being passed as a value.

It's also possible to match the name defined in the RASON model to the SQL parameter name. In this example, the syntax would be

```
cust_data:  { type: "csv",
              connection: "customers.txt",
              selection: "custID = $cuID",
              parameters: {
                 cuID: { binding: 'get', value: 'c1' },
                 marryStat: {binding: 'get', value: 'c2'}
              },
```

Or, with multiple selections…

```
cust_data:  { type: "csv",
              connection: "customers.txt",
                selection: "custID = $cuID and maritalStatus=$marryStat",
              parameters: {
                 cuID: { binding: 'get', value: 'c1' },
                 marryStat: {binding: 'get', value: 'c2'}
              },
```

The remaining data is passed in the data section. The input data `custExist` is passed as a constant within the RASON model.

```
data: {
    comment: "use binding to feed dif. values",
    custExist: { value: false },
    custAge: { value: 40,  binding: 'cust_data', valueCol: 'age' },

    maritalStatus: { value: 's', binding: 'cust_data', valueCol:
    'maritalStatus' },

    employmentStatus: { value: 'selfEmployed', binding: 'cust_data',
    valueCol: 'employmentStatus' },
    creditScore: { value: 610,   binding: 'cust_data', valueCol:
    'creditScore' },
    bankrupt:{ value: false, binding: 'cust_data', valueCol: 'bankrupt' },
    monthIncome:   { value: 2500, binding: 'get' },
    monthExpenses: { value: 1000, binding: 'get' },
    loanType: { value: 'standard', binding: 'loan_data', valueCol: 'type'
    },
    loanRate: { value: 5.0, binding: 'loan_data', valueCol: 'rate' },
    loanTerm: { value: 30,  binding: 'loan_data', valueCol: 'term' },
    loanAmnt: { value: 100000.0, binding: 'get' }
},
```

The decision table results return the recommended loan strategy for customer 1.

```
 {   "loanstrategy": {
      "status" : { "code" : 0, "codeText" : "Solver has
      completed the calculation." },
      "observations" : {
         "strategy" : { "value" : "bureau" },
         "routing" : { "value" : "accept" }
      }
   }
}
```

Changing Table Components Outside of the RASON Model

Input data `monthIncome`, `monthExpenses`, and `loanAmnt` are passed within the RASON model as "get" only. This property allows write access to the data outside of the model environment using the keyword "get". For example, let's say we wanted to increase the value for monthIncome but we did not want to do so within the RASON model. Rather we could pass this new parameter in the call to the "decision" endpoint.

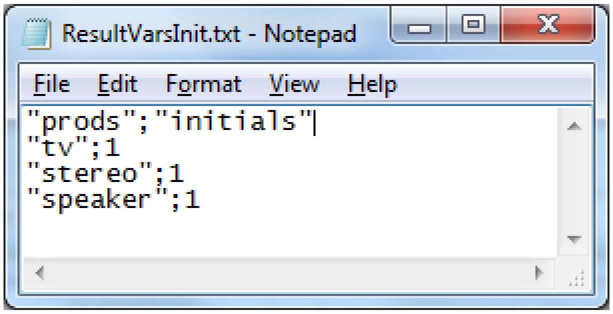
```
$.get(https://rason.net/api/decision?monthIncome=3000...
```

Note: Changing a decision table component outside of the RASON model is not supported.

See the Loan Strategy Example within the RASON User Guide for a complete walkthough of this example.

Exporting

In the example below, initial variable values are first imported from the CSV file ResultVarsInit.txt and, after the model is solved, the final variable values are saved back to that same CSV file. The final constraint values are saved to ResultFcns.txt and the final objective value is saved to ResultObj.txt. (To open and view the complete example file, RGProductMixCsv1.json, and the three files containing the results, ResultVarsInit.txt, ResultFcns.txt, and ResultObj.txt, browse to (typically) C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON.) A screenshot for the file ResultVarsInit.txt is shown below. This file contains the `prods` dimension and the initial variable values. Both ResultFcns.txt and ResultObj.txt will be created once the model is solved.



```
"prods";"initials"
"tv";1
"stereo";1
"speaker";1
```

Note: It is currently not possible to import the initial variable values from one datasourse and export the final variable values to a different data source. In addition, each variable/uncertainVariable block or constraint/uncertainFunction/objective block must be saved to a unique data source.

```
{   datasources : {
      …
      vars_data:   { type: "csv", connection: "ResultVarsInit.txt",
                   indexCols: ['prods'], valueCols: ['initials'],
                   direction: "import/export" },

      fcns_data:   { type: "csv", connection: "ResultFcns.txt", direction:
                    "export"  },

      obj_data:    { type: "csv", connection: "ResultObj.txt", direction:
                   "export"  }
    },
…
 variables : {
      x: { binding: "vars_data", valueCol: 'initials', lower: 0,
      finalValue: [] }
   },
   constraints: {
      c: {dimensions: ['parts'], binding: "fcns_data", formula:
      "MMULT(piv_parts, x)", upper: 'invent', finalValue: [] }
    },
   objective : {
         total: { binding: "obj_data", formula: "sumproduct(x, profit)",
      type: "maximize", finalValue: [] }
   }
```

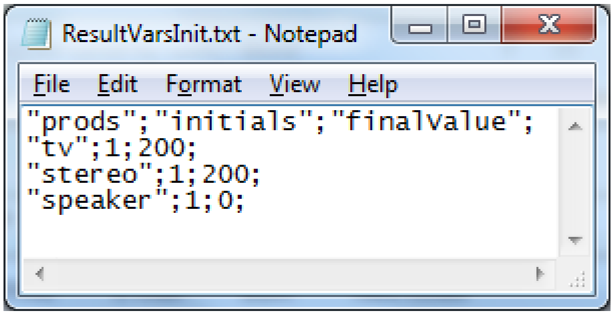
Let's start with the `fcns_data` and `obj_data` data sources. The `fcns_data` datasource exports the final constraint values to the TXT file, ResultFcns.txt. The first property, `type`, specifies the type of file where the data is being imported/exported. In this example, the file is a CSV file.   The second property, `connection`, specifies the file name within quotes (`"ResultFcns.txt"`) while the third property assigns the direction of the file as "export". The `binding` property within the `c` constraint definition "binds" the constraint block to the `fcns_data` data source. The user must specify which results he/she would like exported. In this example, only one output property is passed, `finalValue:[]`. For a complete list of results that may be written to a writeable data source, see the `constraints` section discussion.

Similarly, the `obj_data` datasource exports the final objective value to the TXT file, ResultObj.txt. Again the first property, `type`, specifies the file type (`"CSV")` while the second property, `connection`, specifies the file name (`"ResultObj.txt")` and the third, `direction`, specifies that the file will be exported. Within `objective`, the `total` objective function is bound to the `obj_data` data source. The only output property passed within the `objective` definition is `finalValue:[]`. As a result, only the final objective function value will be exported to ResultObj.txt.   For a complete list of results that may be written to a writeable data source, see the `objective` section discussion.

The `vars_data` data source performs a dual function by first importing the decision variable initial values from the CSV file ResultVarsInit.txt and then saving the final variable values back to that same file. As discussed above, the first property, `type`, specifies the type of file where the data is being imported/exported (`"csv"`) while the second property, `connection`, specifies the file name within quotes (`"ResultVarsInit.txt"`). The third and fourth properties (indexCols and valueCols) are required for importing the initial variable values. The third property, `indexCols` specifies the dimension(s) (or column(s)) to be imported and the fourth property, `valueCols`, specifies the values to be imported. Within `variables`, a block of decision variables `x` is bound to the `vars_data` data source using the `binding` property. Since only one output property is passed within the x definition, `finalValue:[],` a single column containing the final values of the decision variables will be appended to ResultVarsInit.txt. The last property, `direction: "import/export"`, stipulates that the contents of the file will be "imported" (for initial variable values) and then the final variable values will be "exported", hence the setting "import/export".

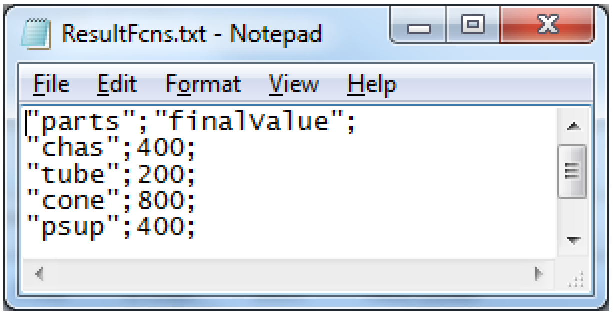
For a complete list of results that may be written to a writeable data source, see the `variables` section discussion.

The export results are shown in the three screenshots below starting with ResultVarsInit.txt. Notice that a new column has been appended, `finalValue`. (This is the result of the `finalValue` output property within the `x` array definition.)

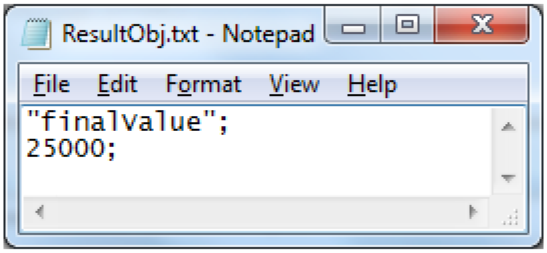


```
"prods";"initials";"finalValue";
"tv";1;200;
"stereo";1;200;
"speaker";1;0;
```

Screenshots of the newly created files ResultFcns.txt and ResultObj.txt are shown below.



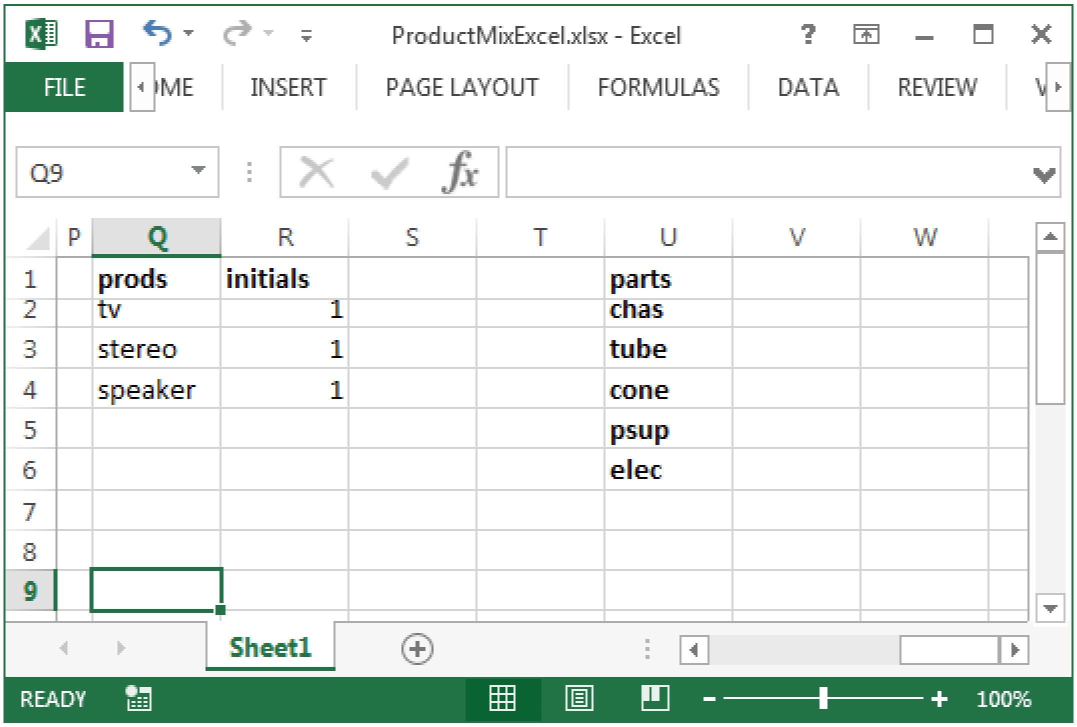
```
"parts";"finalValue";
"chas";400;
"tube";200;
"cone";800;
"psup";400;
```



```
"finalValue";
25000;
```

To perform the same steps when importing/exporting to an Excel file, you need to specify the cell address containing the dimensions to be imported and/or the cells to which the results should be exported. In this second export example (below), the `vars_data` data source is again importing initial decision variable values and then exporting the final decision variable values to the same Excel workbook. A screenshot of

ProductMixExcel.xls is shown below. (To open this file, log on to www.RASON.com, then click the Editor tab and RASON Examples – Example models discussed in RASON Reference Guide.)



	P	Q	R	S	T	U	V	W
1		prods	initials			parts		
2		tv	1			chas		
3		stereo	1			tube		
4		speaker	1			cone		
5						psup		
6						elec		
7								
8								
9								

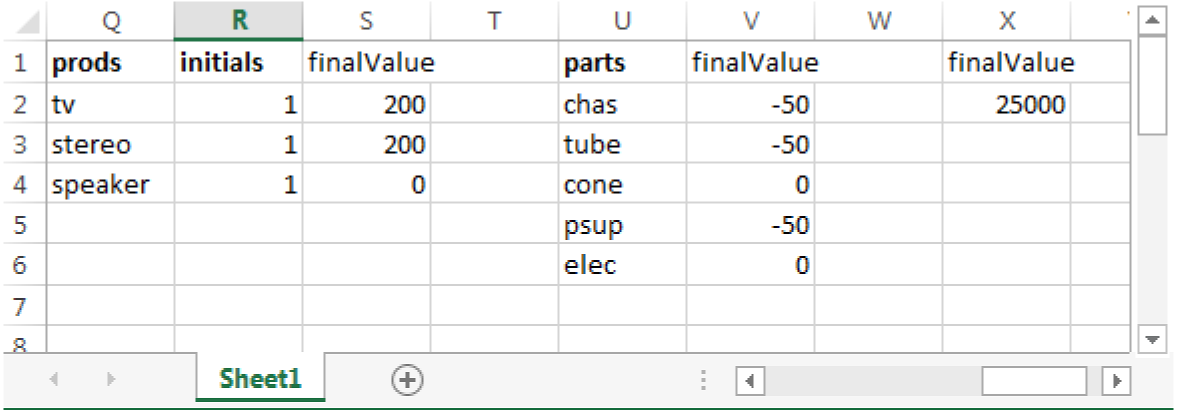
```
datasources: {
   vars_data:   { type: "excel", connection: "ProductMixExcel.xlsx",
   selection: "Sheet1!Q2:R4", indexCols: ['prods'], valueCols:
   ['initials'], direction: "import/export"},
   fcns_data:   { type: "excel", connection: "ProductMixExcel.xlsx",
   selection: "Sheet1!U2:U6", indexCols: ['parts'], direction: "export" },
   obj_data: {type: "excel", connection: "ProductMixExcel.xlsx",
   selection: "Sheet1!X1", direction: "export"}
},
variables : {
   x: { dimensions: ['prods'], binding: "vars_data", valueCol: 'initials',
   lower: 0, finalValue: []},
constraints : {
   c: { dimensions: ['parts'], binding: "fcns_data", formula:
   "MMULT(piv_parts, x) - invent", upper: 0, finalValue: []}
},
objective : {
   total: { formula: "sumproduct(x, profit)", type: "maximize", binding:
   "obj_data", finalValue: [] }
}
```

Again, let's start with the `fcns_data` and `obj_data` data sources. The `fcns_data` datasource exports the final constraint values to the Excel file, ProductMixExcel.xlsx. The first property, `type: "Excel"`, specifies the type of file where the data is being imported/exported, in this instance an Excel file. The second property, `connection`, specifies the file name within quotes (`"ProductMixExce.xlsx"`). The 3rd property, `selection: "Sheet1!U2:U6"`, gives the location, within the ProductMixExcel.xlsx workbook, where the final constraint values will be saved. Alternatively, we could pass a defined name here. The 4th property, `indexCols`, indexes the data by the `"prods"` column (or dimension). The `binding` property within the `c` constraint definition "binds" the constraint block c to the `fcns_data` datasource. One output property (`finalValue:[]`) is included in the `c` definition. The RASON interpreter will append the final constraint values to the original selection. The last property, "direction", indicates that the file will be exported. (The default setting for the `direction` property is "import".) For a complete list of results that may be written to a writeable data source, see the `constraints` section discussion.

Similarly, the `obj_data` datasource exports the final objective value to ProductMixExcel.xlsx. Again the first property, `type`, specifies the file type (`"Excel"`), the second property, `connection`, specifies the file name (`"ResultObj.txt"`) and the third property, `selection`, specifies where the final objective value will be written (`"Sheet1!X1"`). The `binding` property within the `total` objective definition "binds" the objective definition to the `obj_data` data source. Since only the `finalValue` result property is present within the `objective` definition, only the final value of the objective function (a single value) will be saved to ProductMixExcel!X2. The last property, "direction", indicates that the file will be exported. For a complete list of results that may be written to a writeable data source, see the objective section discussion.

As in the example above, the `vars_data` data source performs a dual function by first importing the decision variable initial values from the Excel file ProductMixExcel.xlsx and then saving the final variable values back to that same file. The first property, `type`, specifies the type of file where the data is being imported/exported (`"Excel"`), the second property, `connection`, specifies the file name within quotes (`"ResultVarsInit.txt")` and the third property, `selection`, specifies where the final variable values will be appended (`"Sheet1!Q2:R4").` We could pass a defined name here rather than a cell address. The fourth and fifth properties (`indexCols` and `valueCols`) are required for importing the initial variable values. The property, `indexCols` specifies the dimension(s) (or column(s)) to be imported and the property, `valueCols`, specifies the value column to be imported. Within `variables`, the x array definition uses the `valueCol` property to pass the initial variable values and the `binding` property to "bind" to the `vars_data` data source. Since only one output property (`finalValue:[])` is present within the `x` array definition, only the final variable values will be appended to the original cell address, Q2:R4. The last property, "direction", indicates that the file will be both imported and exported hence the setting "import/export".

The export results are shown in the screenshot below.

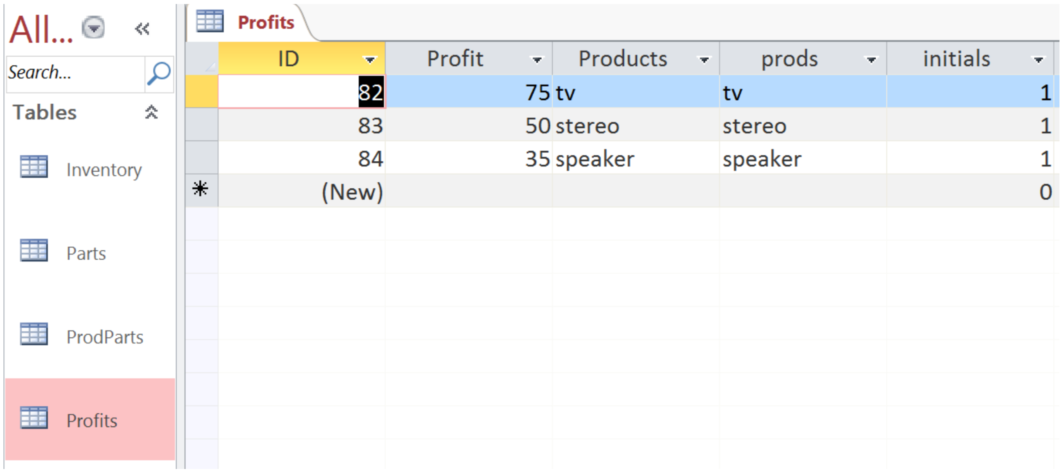


	Q	R	S	T	U	V	W	X
1	prods	initials	finalValue		parts	finalValue		finalValue
2	tv	1	200		chas	-50		25000
3	stereo	1	200		tube	-50		
4	speaker	1	0		cone	0		
5					psup	-50		
6					elec	0		
7								
8								

Note: If the "initials" column heading in Excel is replaced with "finalValue". RASON will both read the initial variable values from cells R2:R4 and overwrite these values with the final variable values after the model is solved.

The next example illustrates how to import and export data from an odbc database, specifically a Microsoft Access database. The screenshot below shows the Profits table containing the "initials" field containing the starting values of the decision variables.

See the Named Data Connections section above for more information on how to create a named data connection where you can maintain your data access credentials in a secure Azure "vault" rather than passing them in the connection property.



ID	Profit	Products	prods	initials
82	75	tv	tv	1
83	50	stereo	stereo	1
84	35	speaker	speaker	1
(New)				0

```
datasources : {
...
     vars_data: { type: "msaccess", connection:
     "ProductMixAccess.accdb",
     selection: "SELECT Products, initials FROM Profits ORDER BY
     ID", indexCols: ['prods'], valueCols: ['initials'],
     direction: "import/export"
     },
     fcns_data: { type: "msaccess", connection:
     "ProductMixAccess.accdb",
     selection: "ResultFcns", direction:"export"
     },
     obj_data: { type: "msaccess", connection:
     "ProductMixAccess.accdb",
     selection: "ResultObj" , direction: "export"
     }
},
...
variables : {
     x: { dimensions: ['prods'], binding: "profit_data",
     valuecol:'initials', lower: 0, finalValue: [] }
     },
     constraints : {
     c: { dimensions: ['parts'], binding: "fcns_data", formula:
     "MMULT(piv_parts, x)", upper: 'invent', finalValue: [] }
     },
objective : {
     total: { binding: "obj_data", formula: "sumproduct(x,
     profit)", type: "maximize", finalValue: []
     }
}
```

Once again, let's start with the `fcns_data` and `obj_data` data sources. The `fcns_data` datasource, exports the final constraint values to the Access data base file, ProductMixAccess.accdb. The first property, `type: "msaccess"`, specifies the type of file where the data is being imported/exported.

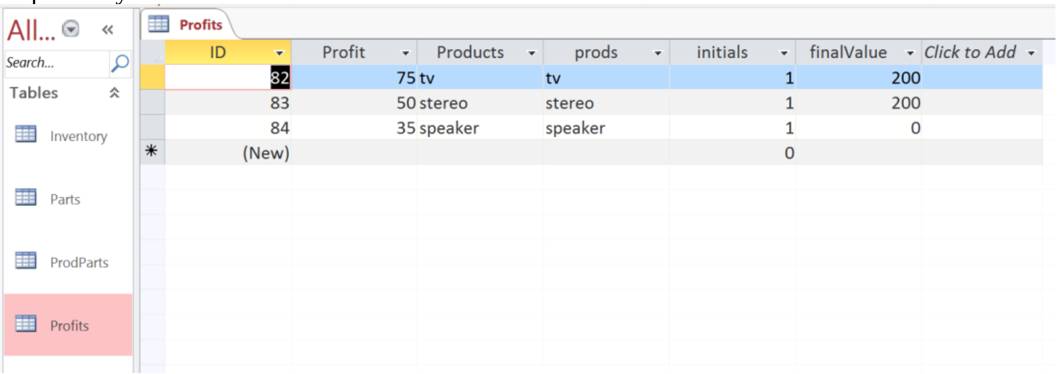
Alternatively, `"access"` or `"odbc"` could have been passed instead of `"msaccess"`. The second property, `connection`, specifies the file name within quotes (`"ProductMixAcess.accdb")`. The 3rd property, `selection: "ResultFcns"`, creates a table within the Access database, where the final constraint values will be saved. The `direction` property ensures that the file is "exported". Within the `c` constraint definition, the constraint block is bound to the `fcns_data` data source by the `binding` property. The output property, `finalValue:[]`, will export the final constraint values to the ResultsFcns table within the file, ProductMixAccess.accdb.

Note: See the Named Data Connections section above for more information on how to create a named data connection where you can maintain your data access credentials in a secure Azure "vault" rather than passing them in the connection property.

Similarly, the `obj_data` datasource exports the final objective value to ProductMixAccess.accdb. Again the first property, `type`, specifies the file type (`"msaccess"`), the second property, `connection`, specifies the file name (`"ProductMixAccess.accdb"`) and the third property, `selection`, specifies the table where the final objective value will be written (`"ResultObj"`). The `direction` property ensures that the file is "exported". The binding property within the `total` objective definition "binds" the objective to the `obj_data` source. The output property, `finalValue:[]`, exports the final objective function value to the ResultObj table within the file ProductMixAccess.accdb.

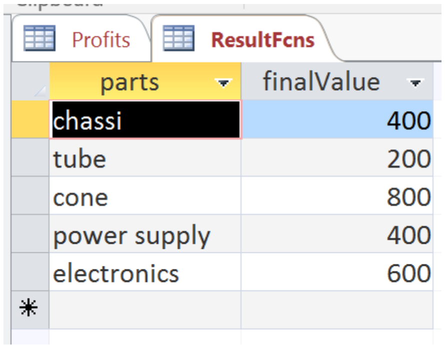
As in the two previous examples, the `vars_data` data source performs a dual function by first importing the decision variable initial values from the Access database and then saving the final variable values back to that same file. The first property, `type`, specifies the type of file where the data is being imported/exported (`"msaccess"`), the second property, `connection`, specifies the file name within quotes (`"ProductMixAccess.accdb"`) and the third property, `selection`, imports two fields from the Profits table in order by ID (`"SELECT Products, Initials FROM Profits ORDER BY ID"`). The 4th property, `indexCols`, indexes the data by the `prods` dimension while the 5th property, `valueCols`, imports the actual numerical data. The `direction` property ensures that the initial variable values are imported and the final variable values are exported using `direction: "import/export.` Within `variables`, the `x` array definition uses the `valueCol` property to pass the initial variable values and the `binding` property to "bind" to the `vars_data` data source. The final variable values (requested using the output property `finalValues:[]` within the `x` array definition) will be appended to the Profits table.

The results of the export are shown in the screenshots below. The first screenshot displays the Profits table. Notice the appended finalValue field. The 2nd and 3rd screenshots display the ResultFcns and ResultObj tables, respectively.

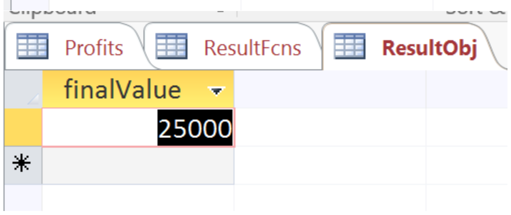


Profits

ID	Profit	Products	prods	initials	finalValue	Click to Add
82	75	tv	tv	1	200	
83	50	stereo	stereo	1	200	
84	35	speaker	speaker	1	0	
(New)				0		



parts	finalValue
chassi	400
tube	200
cone	800
power supply	400
electronics	600



finalValue
25000

Note: See the Rason User Guide to see an example of how to import and export to/from an SQL Database running on an Azure server running in the Cloud using an ODBC connection string.

All properties available for `dataSources`, can be found in the table below.

Data Source Property	Example	Explanation

`Type`	`type: "Excel"`
	`type: "odbc"`
	`type: "csv"`
		Use this property to pass the file type: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "odata" (OData database), "mssql" (Microsoft Sequel), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file).

`connection`	`connection: "ProductMix.xlsx"`
	`connection: "ProductMixCSV.txt;header"`
	Note: See the Named Data Connections section above for more information on how to create a named data connection where you can maintain your data access credentials in a secure Azure "vault" rather than passing them in the connection property.
		Use this property to pass the filename of the data source.
		If using a CSV file with column headings, you must also pass "header", i.e.:

`selection`	1a. `selection: "Sheet1!B2:D6"` 1b. `selection: "Parts_Table"` 2. `Selection: "SELECT Parts, Products, Qty FROM Parts ORDER BY ID"`	Use this property to select the columns/fields to import. 1. If data source is an Excel file, pass A. the Excel Range or B. an Excel defined name. 2. If data source is an odbc database use: `SELECT` + desired fields separated by commas + `FROM` +

		name of table containing desired field(s) + `ORDER BY` + field name containing order index
`indexCols`	`indexCols: ["parts", "prods"]` Note: The properties indexCols and valueCols create a RASON table and should not be used with colIndex or rowIndex which create a dataframe.	Used in conjunction with `valueCols.` Use this property to index by dimension(s).
`valueCols`	`valueCols: ["qty"]` Note: The properties indexCols and valueCols create a RASON table and should not be used with colIndex or rowIndex which create a dataframe.	Used in conjunction with `indexCols`. Use this property to import columns/fields containing values
`colIndex`	`colIndex: "prods"` Note: The properties colIndex and rowIndex create a dataframe and should not be used with indexCols or valueCols which create a RASON table.	Use this property to create an implicit index set consisting of integer numbers from 1 to the number of columns. This property should be used when importing data not organized as a table, and thus not having index columns or value columns.
`rowIndex`	`rowIndex: "parts"` Note: The properties colIndex and rowIndex create a dataframe and should not be used with indexCols or valueCols which create a RASON table.	Use this property to create an implicit index set consisting of integer numbers from 1 to the number of rows. This property should be used when importing data not organized as a table, and thus not having index columns or value columns.
`sortIndexCols` `sort`	`sortIndexCols: True`	Use this property to sort the columns alphabetically.

`direction`	`direction: "import"`
	`direction: "export"`
	`direction: "import/export"`
		Use this property to specify if the contents of the data source are being imported ("import" – the default), exported ("export") or both ("import/export")

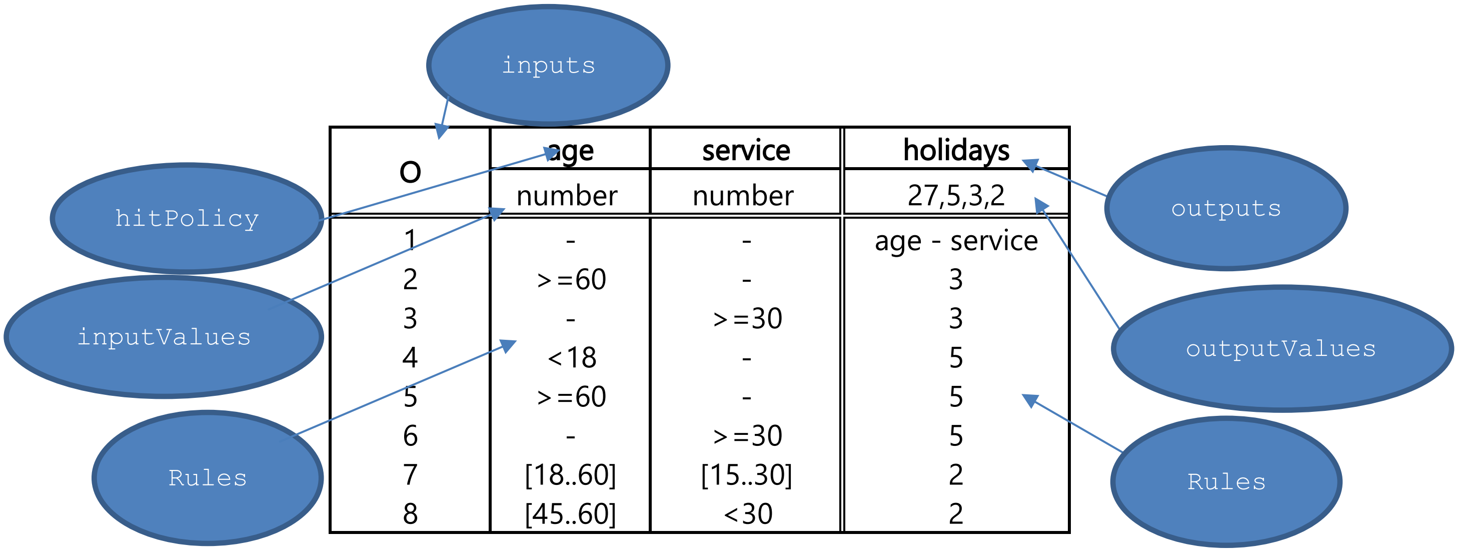
Decision Tables

A decision table contains a set of rules which specify actions to perform based on specific conditions. Decision tables should be used when there is a consistent number of rules, or conditions, to be evaluated followed by a specific set of actions to be performed once a rule, or condition, is met.

A decision table is created in RASON using the newly introduced `decisionTables` RASON section as shown in the example code below, however additional sections such as data, formulas, or dataSource will also be called into play to pass the calculation parameters, get results, and import data respectively. A RASON model creating and invoking a decision table is below. This example code uses `decisionTables`, `data` and `formulas` components to pass the data to the decision table, create the decision table and finally calculate the decision table.

The modeling language used internally for Rason's Decision Table functionality is S-FEEL extended to standard conversion functions in FEEL. For more information on Decision Tables, we invite you to reference the following: DMN Method and Style by Bruce Silver (Cody Cassidy Press, September 28, 20180 and DMN Cookbook by Bruce Silver & Edson Tirelli (Cody-Cassidy Press, April 4, 2018).

Below, you will find a graphical representation of a decision table and the matching RASON code. See the chart for explanations on each decision table component. See the RASON User Guide for a complete walk through of several decision table examples.



O	age	service	holidays
	number	number	27,5,3,2
1	-	-	age - service
2	>=60	-	3
3	-	>=30	3
4	<18	-	5
5	>=60	-	5
6	-	>=30	5
7	[18..60]	[15..30]	2
8	[45..60]	<30	2

```
{ comment: "'O' output order policy example",
  decisionTables: {
     tblHolidays: {
         hitPolicy: 'outputOrder',
         inputs: ['age', 'service'],
         outputs: ['holidays'],
         refTypes: [‘number’, ‘number’, ‘text’],
         outputValues: [27, 5, 3, 2]
         rules: [
            ['-', '-', 22],
            ['>=60', '-', 3],
            ['-', '>=30', 3],
            ['<18', '-', 5],
            ['>=60','-', 5],
            ['-', '>=30', 5],
            ['[18..60]','[15..30]', 2],
            ['[45..60]','<30', 2]
            ],
         default: [1]
         }
      },
            data: {
               age: { value: 58 },
               service: { value: 31 }
            },
            formulas: {
               result: { formula: "tblHolidays(,,age, service)", finalValue: []
}
             }
           }
         }
```

decisionTables

In RASON, tables are defined as objects in the newly introduced section **decisionTables**: { }. Each table component is defined as **component** with a scalar or array value assigned to it. See the RASON User Guide for a complete walk through of how to create and calculate a decision table. See the table below for all components associated with decisionTables.

Data Source Property	Example	Explanation
`default`	`default: [1]`	There are two ways to return a default value for a decision table. 1. Simply use "-" for all unary tests. 2. Use the default component as shown in the example above.
`hitPolicy`	`hitPolicy: 'outputOrder'`	Specifies how the table will be evaluated when multiple rules are applicable and multiple output values are returned. The Hit Policy value identifies the supported policies by a capital letter and an operator, when applicable. The 1st letter of the policy or the whole word may be passed to the `hitPolicy` component. The currently supported Hit policies and their meanings are: **Unique (U):** A unique rule must be successful, or "hit", evaluating to a unique result. If multiple rules are "hit", an error will be returned. **Any (A):** If rules overlap, but point to the same result, that unique result is returned. **Priority(P):** If multiple rules are "hit" and multiple results collected, the result with the highest priority is returned. Priorities are defined by the order in the `outputValues` component. **First (F):** Returns only 1 result. Once a rule is evaluated successfully, or a hit occurs, the search stops. **Rule Order** (**R**) **-** If multiple rules are hit, the collection of results is returned according to the rule order, as specified in `outputValues`.

		Output Order (**O**) If multiple rules are hit, return the collection of results in the priority order as listed for `outputValues`. **Collect** (**C**) – The same as (**R**). However, we may make this policy more specific by adding an operator to it in order to allow aggregation. Note: If aggregating a date, a scalar is returned. If using an operator, output must not be a string, but only a numerical value. **C**+ **-** totals the matched output values **C**< **-** returns the min of the matched output values **C**> **-** returns the max of the matched output values **C**# **-** returns the number of matched output values
`inputs`	`inputs: ['age', 'service']`	The input parameters, or the inputs to the decision table.
`inputValues`	`inputValues: [`	Describes the domain covered by all input entries in the decision table rules. Each input value must relate to a given input parameter. Input Values may be a list of values separated by commas (i.e. 27, 5, 3, 2) or a list of unary tests (i.e. <10, >=20, [18..20]). Both `refValues` and `inputValues/outputValues` may exist within the same RASON model but `inputValues` will override the `refValues` component. Use `refValues` when stipulating the value type accepted by the column and `input/outputValues` when stipulating the domain of the column. When testing a value against a list of values or unary tests, the OR operator is used. A list of values is evaluated as 27 OR 5 OR 3 OR 2. Likewise, the list of unary tests is evaluated as <10 OR >=20. It's possible to negate a list as well. For example, NOT(27, 5, 3, 2) would result in a selection of a record that does not includes 27 OR 5 OR 3 OR 2. Similarly, NOT(<10, >=20) would equate to neither <10 OR >=20 being selected. All input entries in the relevant input column should cover the entered domain, otherwise, an error will be generated indicating that the table is not complete. If an input value does not exist, the completeness test is not performed.
`outputs`	`outputs: ['holidays']`	After calculation, a decision table returns a few selected or all output parameter values in the form of an array. If a decision table has a single output or a single output has been selected, the result will be a scalar value. These input parameters belong to the local scope of this table.
`outputValues`	`outputValues: [27, 5, 3, 2]`	An `outputValue` can be a list of values separated by commas (i.e. 27, 5, 3, 2) listing the priority of returned results. Ia value appears in the table that does not match the output value, an error will be returned.

		In this example, the list "27, 5,3,2" is entered as an output for "holidays". The output values list (27, 5, 3, 2) specifies the priority when returning the results. In other words, the results are to be returned largest to smallest. Both `refValues` and `outputValues` may exist within the same RASON model but `outputValues` will override the `refValues` component. Use `refValues` when stipulating the value type accepted by the column and `input/outputValues` when stipulating the domain of the column. All output entries in the relevant output column should cover the entered domain, otherwise, an error will be generated indicating that the table is not complete. If an output value does not exist, the completeness test is not performed.
`refTypes`	`refTypes: [‘number’, ‘number’, ‘text’],`	Describes the data type for each input. When defining refTypes, all input and output columns must be included. Enter empty strings or null for any input or output column. The output parameter's domain is defined using the `outputValues` component (see below). May coexist with `inputValues/outputValues.` **<u>Data Types</u>** **Boolean:** The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings. **Number:** May be an integer or fraction. **Text:** Any string **Date:** Any valid date, such as 05-05-1964 **Time:** Any valid time **Duration:** There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S represents 1 hour, 2 minutes, and 3 seconds using: • P for "period" • 1D for 1 day • T for "time" • 1H for 1 hour • 2M for 2 minutes and • 3S for 3 seconds.

`rules`	`rules: [`
	`['-', '-', 22],`
	`['>=60', '-', 3],`
	`['-', '>=30', 3],`
	`['<18', '-', 5],`
	`['>=60','-', 5],`
		Rules consists of Input Entries and Output Entries. These entries consist of unary tests which return information (true or false) about the rule. Supported unary tests may have one of the following syntax forms: value(Boolean, number, text, date, time, duration), < value, > value, <= value, >= value, [value..value], (value..value], [value..value), (value..value), "-". Note: Currently, rules *may only* be entered as rows.

	['-', '>=30', 5], ['[18..60]','[15..30]',2], ['[45..60]','<30', 2]],	The first test examines whether the value being tested is equal to the value inside the parenthesis. For example, if the Unary test consists of the single value 'medium', the resulting test would ensure that the variable being tested was equal to 'medium'. The forms, [value..value], (value..value], [value..value) and (value..value), are interval tests. The [ ] operators denote a closed interval while the ( ) operators denote an open interval. The last test, "-" returns TRUE against any value.

Supplying Data To and Calculating the Decision Table

Use the data section to pass the parameters required to evaluate, or calculate, the decision table. Note: Decision tables only accept scalar (constant) arguments as inputs.

```
data: {
      input1: { value: XX },
      input2: { value: YY }
 },
```

Data may be passed directly in the RASON code or from an external data source. See the `dataSource` section in this guide for more information on external data sources. See the "Using an External Data Source" example in the RASON User Guide for a walkthrough of an example where the data is contained within a CSV file. Note: No components of a decision table can be bound to an external database or file.

Use the formulas section to obtain the final results from a decision table given the input data. The complete signature of the decision table function is:

```
formulas: {
result: { formula: "tblDecTable([string ret_output], [bool ret_header],
input1, input2, … inputN)", finalValue: [] }
      }
  }
```

where:

- The first argument [string ret_output] is an optional argument that, when passed, returns only the desired columns in the output. Note this argument must be surrounded by single quotes, 'Header_Name'.
- The second argument [strong ret_header] is an optional Boolean argument that, when True, returns the column headings in the output.
- The third and remaining arguments pass the input parameters to the decision table.

The number of arguments given to the decision table must be equal to the number of Input Parameters. The finalValue[] argument returns the final collection of results.

Optional Arguments

Two optional arguments may be passed to a decision table: output and ret_header.

To return the result for a given output only, pass the output heading in quotes. For example, to only receive the number of holidays, rather than both holidays and the rule, add "holidays" as as the second argument to the existing formula:

```
{
result: { formula: "tblHolidays('holidays',,age, service")", finalValue:
[] }
}
```

In this instance, only the result collection for "holidays" will be returned, 27, 5, 3.

You can pass as many output arguments as needed.

A 2nd optional argument, ret_header, is a Boolean argument, that, if True, returns a header for the result collection.

```
{
result: { formula: "tblHolidays('holidays', True, age, service)",
finalValue: [] }
}
```

In this instance, the result collection will include only the "holidays" output parameter with the header "holidays" as the first element in the collection: "holidays", 27, 4, 3.

To only include the optional ret_header argument instead of both optional arguments, use:

```
{
result: { formula: "tblHolidays(,True, age, service)", finalValue: [] }
}
```

See the Merging Decision Table Results section in the RASON User Guide for an example in practice.

FEEL Expressions

Variables and constants can be combined through operations called literal expressions. Literal expressions in S-FEEL are similar to formulas in Excel and in the RASON modeling language.

The following operators are supported in combination with decision table rules: addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (**). Variables and constants can be combined using only these supported operators and parentheses. An example of an expression is: 2 * age – service where two variables, age and service, and a constant, 2, are linked by two arithmetic operations (* and -). Note that an expression is a FEEL expression, NOT an Excel formula. In this example, the expression "age – service" appears in the first rule where "age" and "service" refer to cells H7 and I7. This expression does not refer to any appearances of "age" or "service" outside of the scope of this table. For more information on supported conversion expressions, please see the list below.

Supported conversion functions

date(string *date*) returns a date serial number, the same as Excel DATEVALUE

date(number *y*, number *m*, number *d*)

time(string *time*) returns a time serial number, the same as Excel TIMEVALUE

time(number *h*, number *m*, number *s*[, number *offset*]) the optional *offset* is duration in seconds, which can be used to model UTC

duration(string *dur*) returns duration in months or seconds depending on format

yearsAndMonthsDuration(string *from_date*, string *to_date*) return difference between two dates as a duration in months

number(string num, string group_sep, string dec_sep) returns a number from a string

string(float *num*) returns a float number as a string

Supported numeric functions are:

ceiling(number *num, [scale]*) rounds up a number

- A number in FEEL is represented as a pair of integers (a,b) where a is a signed 34 digit integer and s is the scale of the number. To specify a numeric value for the *ceiling* function using the optional scale argument, use ceiling(a,b). *Examples: ceiling(1.5) = 2, ceiling(-1.56,1) = -1.5.* [2]

*deci*mal(number *num*, number *decimals*) rounds a number to the given number of decimals

floor(number *num, [scale]*) rounds down a number

- A number in FEEL is represented as a pair of integers (a,b) where a is a signed 34 digit integer and s is the scale of the number. To specify a numeric value for the *floor* function using the optional scale argument, use floor(a,b). *Examples: floor(1.5) = 1, floor(-1.56,1) = -1.6.* [3]

Supported string functions:

substring(string *str*, number *pos*, number num_*chars*)

stringLength(string *str*)

upperCase(string *str*)

lowerCase(string *str*)

substringBefore(string *str*, string *match*)

substringAfter(string *str*, string *match*)

contains(string *str*, string *match*)

startsWith(string *str*, string *match*)

endsWith(string *str*, string *match*)

Supported list functions are:

min(number n1, number n2,…) returns the minimum number

max(number n1, number n2,…)

sum(number n1, number n2,…)

mean(number n1, number n2,…)

and(bool b1, bool b2,…)

or(bool b1, bool b2,…)

stringJoin(list of strings, [separator]) – joins a list of strings divided by a separator.

- The separater can be an empty string. Null elements in the list parameter are ignored. If list is empty, the result is the empty string. If delimiter is null, the string elements are joined without a separator. string join(["a","b","c"], "_and_") = "a_and_b_and_c" string join(["a","b","c"], "") = "abc" string join(["a","b","c"], null) = "abc" string join(["a"], "X") = "a" string join(["a",null,"c"], "X") = "aXc" string join([], "X") = ""[4]

Please refer to the OMG Specification on DMN for more details on these functions

[2] Source: Object Management Group Decision Model and Notation Version 1.4. OMG Document Number: dtc/21-12-01. URL: https://www.omg.org/index.htm

[3] Source: Object Management Group Decision Model and Notation Version 1.4. OMG Document Number: dtc/21-12-01. URL: https://www.omg.org/index.htm

[4] Source: Object Management Group Decision Model and Notation Version 1.4. OMG Document Number: dtc/21-12-01. URL: https://www.omg.org/index.htm

Supported Operators for Models using Date, Time or Duration

See the ***Decision Table Containing Duration*** example in the RASON User Guide for an illustration on using expressions within decision table rules. The following list contains the supported operations for decision tables containing dates, times, or durations.

now() – returns current date and time

today() – returns current date

Date – date = duration

Time – time = duration

Date + duration = date

Date – duration = date

Time + duration[s] = time

Time – duration[s] = time

Duration + duration = duration

Duration – duration = duration

Duration or number * duration = number

Duration / duration or number = number

Duration or number / duration = number

dayOfWeek(FeelDate date) or dayOfWeek(string date) or dayOfWeek(FeelDateTime datetime)*

dayOfYear(FeelDate date) or dayOfYear(string date) or dayOfYear(FeelDateTime datetime)*

monthOfYear(FeelDate date) or monthOfYear (string date) or monthOfYear(FeelDateTime datetime)*

weekOfYear (FeelDate date) or weekOfYear (string date) or weekOfYear (FeelDateTime datetime)*

*See Example Code below.

Comparison operators {=, !=, >, >=, <, <=} are also allowed but only between identical types.

Important Note: Date, Time, and Duration data types are S-FEEL types not recognizable in an Excel formula. As a result, these data of these types, must arrive as string input arguments in the S-FEEL format. Similarly, when a decision table, custom function or custom type returns these types, the latter are formatted as S-FEEL strings before they enter the RASON environment.

Example Code

```
{
    comment: "Example of specific FEEL operations",
    data: {
      D: { value: "'2021-05-31'" }
    },

    formulas: {
      dow: { feelFormula: "dayOfWeek(D)", finalValue: [] },
      doy: { feelFormula: "dayOfYear(D)", finalValue: [] },
      moy: { feelFormula: "monthOfYear(D)", finalValue: [] },
      woy: { feelFormula: "weekOfYear(D)", finalValue: [] }
    }
}
```

Response

```
{
    "status": {
       "code": 0,
        "id": "2590+2021-09-17-16-31-48-224467",
        "codeText": "Solver has completed the calculation."
    },
    "observations": {
        "dow": {
            "value": "Monday"
        },
        "doy": {
            "value": 151
        },
        "moy": {
            "value": "May"
        },
        "woy": {
            "value": 23
        }
    }
}
```

The path operator

One additional operator, the path operator is supported by RASON when used with decision tables containing dates, times or durations.

If a variable in an S-FEEL expression is of type **Date**, the following path operations are defined:

DateVariable.Year extracts the year component from the date value

DateVariable.Month extracts the month component from the date value

DateVariable.Day extracts the day component from the date value

DateVariable.Weekday extracts the weekday component from the date value

For example, Date(“2019-05-05”).Year returns 2019; Date(“2019-05-05”).Weekday returns 7 for “Sunday”.

If a variable in an S-FEEL expression is of type **Time**, the following path operations are defined:

TimeVariable.Hour extracts the hour component from the time value

TimeVariable.Minute extracts the minute component from the time value

TimeVariable.Second extracts the second component from the time value

For example, Time(“18:50:05”).Hour returns 18; Time(“18:50:05”).Minute returns 50.

If a variable in an S-FEEL expression is of type **Duration** expressed in years and months, the following path operations are defined:

DurationVariable.Years extracts the years component from the duration value

DurationVariable.Months extracts the months component from the duration value

For example, Duration(“P1Y2M”).Years returns 1; Duration(“P1Y2M”).Months returns 2.

If a variable in an S-FEEL expression is of type **Duration** expressed in days and time, the following path operations are defined:

DurationVariable.Days extracts the days component from the duration value

DurationVariable.Hours extracts the hours component from the duration value

DurationVariable.Minutes extracts the minutes component from the duration value

DurationVariable.Seconds extracts the seconds component from the duration value

For example, Duration(“P1DT1H20M10S”).Days returns 1; Duration(“P1DT1H20M10S”).Minutes returns 20.

Example of the path operator in practice

2 *ceiling(duration(dtDuration).minutes/duration('PT20M').minutes)

Display Precision

Use the optional high level property "displayPrecision": X, to control the decimal precision in the RASON model results. (The number of decimal digits displayed in the RASON results.) The minimum setting for this property is -1, the default setting, and the maximum setting is 15 decimal digits.

If calling the Quick Solve endpoints POST rason.net/api/solvetype = Optimize, Simulate, Decision or Diagnose or GET/POST rason.net/api/model/{nameorid}/solvetype = Optimize, Simulate, Decision or Diagnose; the default setting is 1.0E-6.

When the Quick Solve endpoint POST rason.net/api/solvetype=datamine or solve or POST rason.net/api/model/{nameorid}/solvetype=datamine or solve is called, the default decimal precision will be used which is 1E-17.

Otherwise, the property value setting will determine the precision returned, i.e. if "displayPrecsion"=10, the result decimal precision will be 10 digits, or 1E-10.

Engine Settings

In this optional section, you'll specify the engine to be used to solve the optimization, simulation optimization or stochastic optimization model and/or set any engine options. If a specific engine is not selected, the RASON server will analyze your model and automatically select the best engine to solve. If solving a linear optimization model, the Standard LP/Quadratic engine will be automatically selected. If solving a smooth, nonlinear optimization model, the Standard LSGRG engine will be selected and if solving a nonsmooth model, the Standard Evolutionary Engine will be selected. If running a simulation, the Risk Solver Engine will be automatically selected.

The first property for `engineSettings`, `engine,` identifies the engine to be used during the solve. To specific a specific engine, you must use the string values below.

To Specify this Engine	Use
Nonlinear GRG Solver	"GRG Nonlinear"
Standard LP/Quadratic Solver	"LP/Quadratic"
Standard Evolutionary	"Evolutionary"
Interval Global Solver	"Interval Global"
Standard SOCP Engine	"SOCP Barrier"

The example code below selects the Standard GRG Nonlinear engine to solve an optimization model, turns on the Multistart parameter, and sets the maximum time limit to 600 seconds.

```
engineSettings: {
        engine: "GRG Nonlinear", multistart: True, maxTime: 600
    },
```

See below for complete descriptions of each available engine option.

*Note: *Unlimited* in the tables below equals the maximum 32-bit integer setting, 2,147,483,647.

Common Engine Options

The following parameters are used to control all Solver engines including additional "plug in" engines such as Gurobi, Xpress, Large Scale GRG, Large Scale LP/QP, etc. For options specific to a "plug in" engine, please see the Frontline Solvers Engines Guide.

Irreducible Infeasible Set – Bounds

Name	iisBounds
Default	0 – Include Bounds
Min	0
Max	1 – No Bounds
Type	Integer

Instead of the full Irreducible Infeasible Set (IIS), which analyzes both the constraints and variable bounds in your model and attempts to eliminate as many of them as possible, you can produce the IIS minus the variable bounds. The IIS minus the variable bounds analyzes only the constraints while keeping the variable bounds in force. This may be sufficient to isolate the source of the infeasibility, but you must take into account the bounds on all of the variables when analyzing the IIS.

Iterations

Name	iterations
Default	Unlimited*
Min	1
Max	Unlimited*
Type	Integer

The value for the number of Iterations determines the maximum number of iterations ("pivots" for the Simplex Solver, or major iterations for the GRG Solver) that a Solver Engine may perform on one problem. A new "Trial Solution" is generated on each iteration. For problems with integer constraints, the Iterations setting determines the maximum number of iterations for any *one* subproblem.

Max Time

Name	maxTime
Default	Unlimited*
Min	1
Max	Unlimited*
Type	Integer

The value for Max Time determines the maximum time in seconds that the Solver Engine will run before it stops. For problems with integer constraints, this is the total time taken to solve all subproblems explored by the Branch & Bound method.

Number of Threads

Name	numThreads
Default	0 – Use Maximum Number of Threads Available
Min	0
Max	Number of cores on machine, currently 2.
Type	Integer

The RASON server uses virtual machines to run your model. Each virtual machine has the ability to create multiple "threads" of execution that can be run on different processor cores. (Currently the number of threads is 2.) You can control the number of cores used for each Problem. The default value of 0 means: "use as many threads as there are processor cores in the machine." (The actual number of threads used may vary dynamically during execution.) You can instead set this to a specific number of threads.

Precision

Name	precision
Default	1.0e-6
Min	1.0e-4
Max	1.0e-9
Type	Double

The number entered here determines how closely the calculated values of the constraint left hand sides must match the right hand sides in order for the constraint to be satisfied. (This option is not used with the LP/Quadratic or SOCP engines.) With the default setting of 1.0E-6 (0.000001), a calculated left hand side of -1.0E-7 (0.0000001) would satisfy a constraint such as A1 >= 0. Use caution in making this number much smaller, since the finite precision of computer arithmetic virtually ensures that the calculated values will differ from the expected or "true" values by a small amount. On the other hand, setting the Precision to a much larger value would cause constraints to be satisfied too easily. If your constraints are not being satisfied because the values you are calculating are very large (say in millions or billions of dollars), consider adjusting your formulas and data to work in *units of millions,* or setting the "Scaling" parameter (see explanation below) instead of altering the Precision setting. Generally, this setting should be kept in the range from 1.0E-6 (0.000001) to 1.0E-4 (0.0001).

Precision and Integer Constraints

Another use of Precision is determining whether an integer constraint, such as A1:A5 = integer, A1:A5 = binary or A1:A5 = alldifferent, is satisfied. If the difference between the decision variable’s value and the closest integer value is less than the Precision setting, the variable value is treated as an integer.

Use Automatic Scaling

Name	scaling
Default	0 – On
Min	-1 – Off
Max	1 – On
Type	Integer

When Automatic Scaling is turned on (set to 0 or 1), the Solver will attempt to scale the values of the objective and constraint functions internally in order to minimize the effects of a poorly scaled model. A *poorly scaled* model is one that computes values of the objective, constraints, or intermediate results that differ by several orders of magnitude. Poorly scaled models may cause difficulty for both linear and nonlinear solution algorithms, due to the effects of finite precision computer arithmetic.

If your model is nonlinear and you turn on Automatic Scaling, *make sure that the initial values for the decision variables are "reasonable,"* i.e. of roughly the same magnitudes that you expect for those variables at the optimal solution. The effectiveness of the Automatic Scaling option depends on how well these starting values reflect the values encountered during the solution process.

LP/Quadratic Solver Options

If the LP/Quadratic Solver is manually or automatically selected to solve the problem, the parameters for the engine include the parameters in the Common Options section above as well as the parameters described below. Note that the default values for Primal Tolerance and Dual Tolerance have been chosen very carefully; the LP/Quadratic Solver is designed to solve the vast majority of LP problems ‘out of the box’ with these default tolerances.

Derivatives for the Quadratic Solver

Name	derivatives
Default	1 – Forward
Min	1
Max	2 – Central
Type	Integer

When a quadratic programming (QP) problem – one with a quadratic objective and all linear constraints – is solved with the LP/Quadratic Solver, the quadratic Solver extension requires first or second partial derivatives of the objective function at various points. In the LP/Quadratic engine, these derivatives are computed via *finite differencing* and the method used for finite differencing is determined by the setting of the Derivatives parameter. *Forward* differencing uses the point from the previous iteration – where the problem function values are already known – in conjunction with the current point. *Central* differencing relies only on the current point, and perturbs the decision variables in opposite directions from that point. For QP problems, the Central differencing choice yields essentially exact (rather than approximate) derivative values, which can improve solution accuracy and reduce the total number of iterations; however the initial computation of derivatives may take up to twice as long as with Forward differencing.

Do Presolve

Name	presolve
Default	1 – On
Min	0 – Off
Max	1
Type	Integer

When this parameter is set to 1 (which is the default setting), the LP/Quadratic Solver performs a Presolve step before applying the Primal or Dual Simplex method. Presolving often reduces the size of an LP problem by detecting singleton rows and columns, removing fixed variables and redundant constraints, and tightening bounds.

Dual Tolerance

Name	dualTolerance
Default	1.0e-7

Min	0
Max	1.0
Type	Double

The Dual Tolerance is the maximum amount by which the dual constraints can be violated and still be considered feasible. The default values of 1.0E-7 (0.000001) is suitable for most problems.

Primal Tolerance

Name	primalTolerance
Default	1.0e-7
Min	0
Max	1.0
Type	Double

The Primal Tolerance is the maximum amount by which the primal constraints can be violated and still be considered feasible. The default value of 1.0E-7 (0.000001) is suitable for most problems.

Solve Without Integer Constraints

Name	solveWithout
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

If you solve your problem with this parameter set to 1, LP/Quadratic *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem.

LP/Quadratic Solver MIP Parameters

This section describes the parameters which control the LP/Quadratic engine when solving a mixed integer problem. This engine contains an extensive set of options to improve performance on problems that contain integers.

Cuts

Name	cuts
Default	1
Min	1
Max	3
Type	Integer

1 – Automatic, 2 – None, 3 - Aggressive

The LP/Quadratic Solver supports a wide range of cuts. A *cut* is an automatically generated linear constraint for the problem, in addition to the constraints that you specify. This constraint is constructed so that it "cuts off" some portion of the feasible region of an LP subproblem, without eliminating any possible integer solutions. Cuts require more work on each subproblem, but they can often lead more quickly to integer solutions and greatly reduce the number of subproblems that must be explored.

The LP/Quadratic engine employs a wide range of cuts including, Clique, Flow Cover, Gomory, Knapsack, Local Tree, Mixed Integer Rounding, Probing, Two Mixed Integer Rounding, Reduce and Split, and Special Ordered Sets.

When the Cuts parameter is set to 1, the LP/Quadratic engine will automatically determine the best cuts to use on the problem. If the parameter is set to 2, then no cuts will be performed and if the parameter is set to 3, then the LP/Quadratic Solver will apply the most aggressive forms of cuts.

Heuristics

Name	heuristics
Default	1
Min	1
Max	3
Type	Integer

1 – Automatic, 2 – None, 3 - Aggressive

A *heuristic* is a strategy that often – but not always – will find a reasonably good "incumbent" or feasible integer solution early in the search. Heuristics require more work on each subproblem, but they can often lead more quickly to integer solutions and greatly reduce the number of subproblems that must be explored. The LP/Quadratic engine employs several heuristics including: Feasibility Pump, Greedy Cover, Local Search, and Rounding.

When the Heuristics parameter is set to 1, the LP/Quadratic engine will automatically determine the best heuristics to apply to the problem. If the parameter is set to 2, then no heuristics will be used and if the parameter is set to 3, then the LP/Quadratic Solver will use heuristics extensively in trying to find a good initial incumbent.

Integer Cutoff

Name	intCutoff
Default	2.0e+30
Min	-2.0e-30
Max	2.0e+30
Type	Double

This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a previous run of the same or a very similar problem – you can enter this objective value for the Integer Cutoff parameter. This allows the Branch & Bound process to *start* with an "incumbent" objective value (as discussed above under Integer Tolerance) and avoid the work of solving subproblems whose objective can be no better than this value. If you enter a value for this parameter, you must be *sure* that there is an integer solution with an objective value at least this good: A value that is too large (for maximization problems) or too small (for minimization) may cause LP/Quadratic to skip solving the subproblem that would yield the optimal integer solution.

Integer Tolerance

Name	intTolerance
Default	0
Min	0
Max	1.0
Type	Integer

When you solve an integer programming problem, it often happens that the Branch & Bound method will find a good solution fairly quickly, but will require a great deal of computing time to find (or verify that it has found) the optimal integer solution. The Integer Tolerance setting may be used to tell the Solver to stop if the best solution it has found so far is "close enough."

The Branch & Bound process starts by finding the optimal solution without considering the integer constraints (this is called the *relaxation* of the integer programming problem). The objective value of the relaxation forms the initial "best bound" on the objective of the optimal *integer* solution, which can be no better than this. During the optimization process, the Branch & Bound method finds "candidate" integer solutions, and it keeps the best solution so far as the "incumbent." By eliminating alternatives as its proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver engine finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

```
Objective of incumbent - Objective of best bound
------------------------------------------------
              Objective of best bound
```

If the absolute value of this maximum percentage difference is equal to or less than the Integer Tolerance, the Solver will stop and report the current integer solution as the optimal result, with the status, *Solver found an integer solution within tolerance*. If you set the Integer Tolerance to zero, the Solver will "prove optimality" by continuing to search until all alternatives have been explored and the optimal integer solution has been found. This could take a great deal of computing time.

Max Feasible (Integer) Solutions

Name	maxFeasibleSols
	maxIntegerSols
Default	Unlimited*
Min	1
Max	Unlimited*
Type	Integer

The value for the Max Feasible Sols parameter places a limit on the number of feasible integer solutions found by the Branch & Bound algorithm before LP/Quadratic stops with the status result, *The maximum number of integer candidate/feasible solutions was found*. Each feasible integer solution satisfies all of the constraints, including the integer constraints; the Solver retains the integer solution with the best objective value so far, called the "incumbent."

It is entirely possible that, in the process of exploring various subproblems with different bounds on the variables, the Branch & Bound algorithm may find the same feasible integer solution (set of values for the decision variables) more than once; the Max Feasible Solutions limit applies to the total number of integer solutions found, not the number of "distinct" integer solutions.

Max Subproblems

Name	maxSubproblems
Default	Unlimited*
Min	1
Max	Unlimited*
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before LP/Quadratic stops with the status result, *The maximum number of subproblems was reached*. Each subproblem is a "regular" Solver problem with additional bounds on the variables. In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit.

Preprocessing

Name	preprocessing
Default	1
Min	1
Max	3
Type	Integer

1 – Automatic, 2 – None, 3 - Aggressive

When this parameter is set, the LP/Quadratic performs preprocessing on constraints involving integer variables, to simplify the problem and speed up the solution process. Based on the settings of certain binary integer variables, preprocessing can fix the values of other binary integer variables, tighten the bounds on continuous variables, and in some cases, determine that the subproblem is infeasible, so it is unnecessary to solve it at all.

When the Preprocessing parameter is set to 1, the LP/Quadratic engine will automatically determine the best preprocessing features to use on the problem. If the parameter is set to 2, then no preprocessing will be performed and if the parameter is set to 3, then the LP/Quadratic Solver will apply the most aggressive forms of preprocessing.

Solve Without Integer Constraints

Name	solveWithout
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

If you solve your Problem with this parameter set to 1, LP/Quadratic *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem.

SOCP Barrier Solver Parameters

If the SOCP Barrier Solver is selected as the engine to solve the problem, all parameters described above in the Common Parameters section as well as the parameters described below (Precision, Gap Tolerance, Step Size Factor, Feasibility Tolerance, and the Search Direction option group) will be available to the User.

Feasibility Tolerance

Name	feasibilityTolerance
Default	1.0e-6
Min	0
Max	1.0
Type	Double

The SOCP Barrier Solver considers a solution feasible if the constraints are satisfied to within this tolerance.

Gap Tolerance

Name	gapTolerance
Default	1.0e-6
Min	0
Max	1.0
Type	Integer

The SOCP Barrier Solver uses a primal-dual method that computes new objective values for the primal problem and the dual problem at each iteration. When the gap or difference between these two objective values is less than the Gap Tolerance, the SOCP Barrier Solver will stop and declare the current solution optimal.

Power Index

Name	powerIndex
Default	1
Min	0
Max	Unlimited*
Type	Integer

This parameter is used to select a specific search direction when the Search Direction is computed via the Power Class or Power Class with Predictor-Corrector methods.

Search Direction

Name	searchDirection
Default	3
Min	0
Max	3
Type	Integer

The SOCP Barrier Solver offers four options for computing the search direction on each iteration: **0-Power Class, 1-Power Class with Predictor-Corrector, 2-Dual Scaling, and 3- Dual Scaling with Predictor-Corrector.**

Power Class

This option uses the *power class*, which is a subclass of the commutative class of search directions over symmetric cones with the property that the long-step barrier algorithm using this class has polynomial complexity.

Power Class with Predictor-Corrector

This option uses the *power class* as described above, plus a predictor-corrector term.

Dual Scaling

This option uses HKM (Helmberg, Kojima and Monteiro) dual scaling, a Newton direction found from the linearization of a symmetrized version of the optimality conditions.

Dual Scaling with Predictor-Corrector

This option uses HKM dual scaling, plus a predictor-corrector term.

Step Size Factor

Name	stepSizeFactor
Default	0.99
Min	0
Max	1.0
Type	Integer

This parameter is the relative size (between 0 and 1) of the step that the SOCP Barrier Solver may take towards the constraint boundary at each iteration.

Large Scale GRG Nonlinear Solver Parameters

If the GRG Nonlinear Solver is manually or automatically selected as the engine to solve the problem, all parameters described above in the Common Parameters section as well as the parameters described in this section will be available to the User. The Global Optimization options group and the Population Size parameters are described in the next section, "Multistart Search Options."

The default choices for these options are suitable for the vast majority of problems; although it generally won't *hurt* to change these options, you should first consider other alternatives such as improved scaling before attempting to fine-tune them. In some scientific and engineering applications, alternative choices may improve the solution process.

Convergence

Name	convergence
Default	0.0001
Min	0
Max	1
Type	Double

The LSGRG Solver will stop with the status *Solver has converged to the current solution.* when the objective function value is changing very slowly for the last few iterations or trial solutions. More precisely, the LSGRG Solver stops if the absolute value of the *relative* change in the objective function is less than the value in the Convergence edit box for the last 5 iterations. While the default value of 1.0E-4 (0.0001) is suitable for most problems, it may be too large for some models, causing the LSGRG Solver to stop prematurely when this test is satisfied, instead of continuing for more Trial Solutions until the optimality (KKT) conditions are satisfied.

If you are getting this message when you are seeking a locally optimal solution, you can change the parameter setting to a smaller value such as 1.0E-5 (0.00001) or 1.0E-6 (0.000001); but you should also consider why it is that the objective function is changing so slowly. Perhaps you can add constraints or use different starting values for the variables, so that the Solver does not get "trapped" in a region of slow improvement.

Relax Bounds on Variables

Name	relaxBounds
Default	0 – Do Not Relax
Min	0
Max	1 – Relax
Type	Integer

By default, the LSGRG Solver ensures that any trial points evaluated during the solution process will not have values that violate the bounds on the variables you specify, even by a small amount. If your problem functions cannot be evaluated for values outside the variable bounds, this default behavior will ensure that the solution process can continue. However, at times, the LSGRG Solver can make more rapid progress along a given search direction by testing trial points with values slightly outside the bounds on the variables. If you want to permit this to happen, set this option to 1. If you receive the Result, *Float error status. (Internal float error.)*, as a first step you should set this option back to the default, 0.

Step Size

Name	stepSize
Default	1.0e-6
Min	1.0e-9
Max	1.0e-4
Type	Double

This parameter determines the "delta" or amount of change used in computing numerical derivatives via finite differencing. Changing the step size to a very small value could result in the GRG engine stopping prematurely. However, a large step size could result in Solver "stepping over" a good solution. **For the vast majority of models, this parameter should be left at the default value.**

Derivatives and Other Nonlinear Options

The default values for the Estimates, Derivatives and Search options can be used for most problems. If you'd like to change these options to improve performance on your model, this section will provide some general background on how they are used by the LSGRG Solver.

On each major iteration, the LSGRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). The Derivatives option is concerned with how these partial derivatives are computed.

The LSGRG (Generalized Reduced Gradient) solution algorithm proceeds by first "reducing" the problem to an unconstrained optimization problem, by solving a set of nonlinear equations for certain variables (the "basic" variables) in terms of others (the "nonbasic" variables). Then a search direction (a vector in *n*-space, where *n* is the number of nonbasic variables) is chosen along which an improvement in the objective function will be sought. The Search option is concerned with how this search direction is determined.

Once a search direction is chosen, a one-dimensional "line search" is carried out along that direction, varying a step size in an effort to improve the reduced objective. The Step Size parameter controls the size of this step. The initial estimates for values of the variables that are being varied have a significant impact on the effectiveness of the search. The Estimates option is concerned with how these estimates are obtained.

Derivatives

Name	derivatives
Default	1
Min	1
Max	2
Type	Integer

1 – Forward, 2 –Central

On each major iteration, the LSGRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). In the Large Scale GRG Engine, the method used for finite differencing is determined by the Derivatives setting.

Forward differencing (the default choice) uses the point from the previous iteration – where the problem function values are already known – in conjunction with the current point. *Central* differencing relies only on the current point, and perturbs the decision variables in opposite directions from that point. This requires up to twice as much time *on each iteration*, but it may result in a better choice of search direction when the derivatives are rapidly changing, and hence fewer total iterations.

Estimates

Name	estimates
Default	0 – Tangent
Min	0
Max	1 – Quadratic
Type	Integer

This option determines the approach used to obtain initial estimates of the basic variable values at the outset of each one-dimensional search. The Tangent choice uses linear extrapolation from the line tangent to the reduced objective function. The Quadratic choice extrapolates the minimum (or maximum) of a quadratic fitted to the function at its current point. If the current reduced objective is well modeled by a quadratic, then the Quadratic option can save time by choosing a better initial point, which requires fewer subsequent steps in each line search. If you have no special information about the behavior of this function, the Tangent choice is "slower but surer." **Note:** the Quadratic choice here has no bearing on quadratic programming problems.

Search Option

Name	searchOption
Default	0 – Newton
Min	0
Max	1 – Conjugate
Type	Integer

It would be expensive to determine a search direction using the pure form of Newton’s method, by computing the Hessian matrix of *second* partial derivatives of the problem functions. Instead, a direction is chosen through an estimation method. The default choice, Newton, uses a quasi-Newton (or BFGS) method, which maintains an *approximation* to the Hessian matrix; this requires more storage (an amount proportional to the square of the number of currently binding constraints) but performs very well in practice. The alternative choice, Conjugate, uses a conjugate gradient method, which does not require storage for the Hessian matrix and still performs well in most cases. The choice you make here is not crucial, since the LSGRG solver is capable of switching *automatically* between the quasi-Newton and conjugate gradient methods depending on the available storage.

Multistart Search Parameters

This section discusses the Global Optimization options group and the Population Size parameters that are used by the Large Scale GRG Solver Engine. For reproducible results when using the Multistart Search Parameters, use the model option, Random Seed. (See the Engine Settings section for details.)

These parameters control the multistart methods for global optimization, which will automatically run the GRG Solver (or certain field-installable Solver engines) from a number of starting points in order to seek the globally optimal solution.

Multistart Search

Name	multistart
Default	0 – Off
Min	0
Max	1 – On

Type	Integer

If this parameter is set to 1, the multistart methods are used to seek a globally optimal solution. If this parameter is set to 0, the other options described in this section are ignored. The multistart methods will generate candidate starting points for the GRG Solver (with randomly selected values between the bounds you specify for the variables), group them into "clusters" using a method called multi-level single linkage, and then run the GRG Solver from a representative point in each cluster. This process continues with successively smaller clusters that are increasingly likely to capture each possible locally optimal solution.

Population Size

Name	populationSize
Default	0
Min	0
Max	200
Type	Integer

The multistart methods generate a number of candidate starting points for the GRG Solver equal to the value that you enter for the parameter. This set of starting points is referred to as a "population," because it plays a role somewhat similar to the population of candidate solutions maintained by the Evolutionary Solver. The minimum population size is 10 points; if you supply a value less than 10, or do not pass the parameter at all, the multistart methods use a population size of 10 times the number of decision variables in the problem, but no more than 200.

Require Bounds on Variables

Name	requireBounds
Default	1 – On
Min	0 - Off
Max	1
Type	Integer

This parameter is turned on, set to 1, by default, but it comes into play only when the Multistart Search box is checked. The multistart methods generate candidate starting points for the GRG Solver by randomly sampling values between the bounds on the variables that you specify. If you do not specify both upper and lower bounds on each of the decision variables, the multistart methods can still be used, but because the random sample must be drawn from an "infinite" range of values, this is unlikely to effectively cover the possible starting points (and therefore have a good chance of finding all of the locally optimal solutions), unless the GRG Solver is run on a great many subproblems, which will take a very long time.

The tighter the bounds on the variables that you can specify, the better the multistart methods are likely to perform. (This is also true of the Evolutionary Solver.) Hence, this option is turned on by default, so that you will be automatically reminded to include both upper and lower bounds on all of the variables whenever you select Multistart Search. If both the Multistart Search and Require Bounds on Variables parameters are both set to 1, but you have not defined upper and lower bounds on all of the variables, you will receive the Status result, *Missing bounds status. Returned for EV/MSL Require Bounds when bounds are missing.*

If you get this result, you must either add both upper and lower variable bounds or else set the Require Bounds parameter to 0 and resolve.

Topographic Search

Name	topoSearch

Default	0 – Off
Min	0
Max	1 – On
Type	Integer

If this parameter (and the Multistart parameter) are both set to 1, the multistart methods will make use of a "topographic" search method. This method uses the objective value computed for the randomly sampled starting points to compute a "topography" of overall "hills" and "valleys" in the search space, in an effort to find better clusters and start the GRG Solver from an improved point (already in a "hill" or "valley") in each cluster. Computing the topography takes extra time, but on some problems this is more than offset by reduced time taken by the GRG Solver on each subproblem.

Evolutionary Solver Parameters

If the Evolutionary Solver is selected as the engine to solve the problem, all parameters described above in the Common Parameters section as well as the parameters described in this section will be available to the User.

As with the other Solver engines, the Max Time option determines the maximum amount of time the Evolutionary Solver will run before stopping with the status result, *Time out status. Returned when the maximum allowed time has been exceeded. Indicates an early exit of the algorithm.* The Iterations option rarely comes into play, because the Evolutionary Solver always uses the Max Subproblems and Max Feasible Solutions parameters, whether or not the problem includes integer constraints. (The count of iterations is reset on each new subproblem, so the Iterations limit normally is not reached.) The Precision option plays the same role as it does in the other Solver engines – governing how close a constraint value must be to its bound to be considered satisfied, and how close to an exact integer value a variable must be to satisfy an integer constraint. It also is used in computing the "penalty" applied to infeasible solutions that are accepted into the population: A smaller Precision value increases this penalty.

Convergence

Name	convergence
Default	0.0001
Min	0
Max	1
Type	Double

The Evolutionary Solver will stop with the Status result, *The Solver has converged to the current solution,* if nearly all members of the current population of solutions have very similar "fitness" values. Since the population may include members representing infeasible solutions, each "fitness" value is a combination of an objective function value and a penalty for infeasibility. Since the population is initialized with trial solutions that are largely chosen at random, the comparison begins after the Solver has found a certain minimum number of improved solutions that were generated by the evolutionary process. The stopping condition is satisfied if 99% of the population members all have fitness values that are within the Convergence tolerance of each other.

If you believe that the engine is stopping prematurely with the status *The Solver has converged to the current solution*, you can make the Convergence tolerance smaller, but you may also want to increase the Mutation Rate and/or the Population Size, in order to increase the diversity of the population of trial solutions.

Fix Nonsmooth Variables

Name	fixNonsmooth
Default	0 – Off
Min	0

Max	1 – On
Type	Integer

This parameter determines how non-smooth variable occurrences in the problem will be handled during the local search step. If this parameter is set to 1, the non-smooth variables are fixed to their current values (determined by genetic algorithm methods) when a nonlinear Local Gradient or linear Local Gradient search is performed; only the smooth and linear variables are allowed to vary. If this parameter is set to 0, all of the variables are allowed to vary.

Since gradients are undefined for non-smooth variables at certain points, fixing these variables ensures that gradient values used in the local search process will be valid. On the other hand, gradients *are* defined for non-smooth variables at *most* points, and the search methods are often able to proceed in spite of *some* invalid gradient values, so it often makes sense to vary all of the variables during the search. Hence, this parameter is set to 0 by default; you can experiment with its setting on your model.

Local Search

Name	localSearch
Default	3
Min	0
Max	3
Type	Integer

0-Randomized Local Search, 1- Deterministic Pattern Search, 2-Gradient Local Search, 3-Automatic Choice

This option determines the local search strategy employed by the Evolutionary Solver. As noted under the Mutation rate option, a "generation" or subproblem in the Evolutionary Solver consists of a possible mutation step, a crossover step, an optional local search in the vicinity of a newly discovered "best" solution, and a selection step where a relatively "unfit" member of the population is eliminated. You have a choice of strategies for the local search step. You can use Automatic Choice (the default), which selects an appropriate local search strategy automatically based on characteristics of the problem functions.

Randomized Local Search

This local search strategy generates a small number of new trial points in the vicinity of the just-discovered "best" solution, using a probability distribution for each variable whose parameters are a function of the best and worst members of the current population. (If the generated points do not satisfy all of the constraints, a variety of strategies may be employed to transform them into feasible solutions.) Improved points are accepted into the population.

Deterministic Pattern Search

This local search strategy uses a "pattern search" method to seek improved points in the vicinity of the just-discovered "best" solution. The pattern search method is deterministic – it does not make use of random sampling or choices – but it also does not rely on gradient information, so it is effective for non-smooth functions. It uses a "slow progress" test to decide when to halt the local search. An improved point, if found, is accepted into the population.

Gradient Local Search

This local search strategy makes the assumption that the objective function – even if non-smooth – can be approximated locally by a quadratic model. It uses a classical quasi-Newton method to seek improved points, starting from the just-discovered "best" solution and moving in the direction of the gradient of the objective function. It uses a classical optimality test and a "slow progress" test to decide when to halt the local search. An improved point, if found, is accepted into the population.

Automatic Choice

This option allows the Solver to select the local search strategy automatically in the engine. The Solver uses diagnostic information from the Polymorphic Spreadsheet Interpreter to select a linear Gradient Local Search strategy if the problem has a mix of non-smooth and linear variables, or a nonlinear Gradient Local Search strategy if the objective function has a mix of non-smooth and smooth nonlinear variables. It also makes limited use of the Randomized Local Search strategy to increase diversity of the points found by the local search step.

Mutation Rate

Name	mutationRate
Default	0.075
Min	0
Max	1.0
Type	Double

The Mutation Rate is the probability that some member of the population will be mutated to create a new trial solution (which becomes a candidate for inclusion in the population, depending on its fitness) during each "generation" or subproblem considered by the evolutionary algorithm. In the Evolutionary Solver, a subproblem consists of a possible mutation step, a crossover step, an optional local search in the vicinity of a newly discovered "best" solution, and a selection step where a relatively "unfit" member of the population is eliminated.

There are many possible ways to mutate a member of the population, and the Evolutionary Solver actually employs five different mutation strategies, including "permutation-preserving" mutation strategies for variables that are members of an "alldifferent" group. The Mutation Rate is effectively subdivided between these strategies, so increasing or decreasing the Mutation Rate affects the probability that each of the strategies will be used during a given "generation" or subproblem.

Population Size

Name	populationSize
Default	0
Min	0
Max	200
Type	Integer

The Evolutionary Solver maintains a population of candidate solutions, rather than a "single best solution" so far, throughout the solution process. This option sets the number of candidate solutions in the population. The minimum population size is 10 members; if you supply a value less than 10 for this option, or not pass anything for the population, the Evolutionary Solver uses a population size of 10 times the number of decision variables in the problem, but no more than 200.

The initial population consists of candidate solutions chosen largely at random, but it always includes at least one instance of the starting values of the variables (adjusted if necessary to satisfy the bounds on the variables), and it may include more than one instance of the starting values, especially if the population is large and the initial values represent a feasible solution.

A larger population size may allow for a more complete exploration of the "search space" of possible solutions, especially if the rate is high enough to create diversity in the population. However, experience with genetic and evolutionary algorithms reported in the research literature suggests that a population need not be very large to be effective – many successful applications have used a population of 70 to 100 members.

Require Bounds on Variables

Name	requireBounds
Default	1 – On
Min	0 – Off
Max	1
Type	Integer

If the parameter "Require Bounds on Variables" is set to 1, and some of the decision variables do not have upper or lower bounds specified at solve time, the engine will stop immediately with the Status result, *Missing bounds status. Returned for EV/MSL Require Bounds when bounds are missing*. If this parameter is set to 0, the Solver will not require upper and lower bounds on the variables, but will attempt to solve the problem without them. Note that this parameter is turned on by *default*.

Bounds on the variables are especially important to the performance of the Evolutionary Solver. For example, the initial population of candidate solutions is created, in part, by selecting values at random from the ranges determined by each variable's lower and upper bounds. Bounds on the variables are also used in the mutation process – where a change is made to a variable value in some member of the existing population – and in several other ways in the Evolutionary Solver. If you do not specify lower and upper bounds for all of the variables in your problem, the Evolutionary Solver can still proceed, but the almost-infinite range for these variables may significantly slow down the solution process, and make it much harder to find "good" solutions. Hence, it pays for you to determine realistic lower and upper bounds for the variables.

Filtered Local Search

The Evolutionary engine applies two tests or "filters" to determine whether to perform a local search each time a new point generated by the genetic algorithm methods is accepted into the population. The "merit filter" requires that the objective value of the new point be better than a certain threshold if it is to be used as a starting point for a local search; the threshold is based on the best objective value found so far, but is adjusted dynamically as the engine proceeds. The "distance filter" requires that the new point's distance from any known locally optimal point (found on a previous local search) be greater than the distance traveled when that locally optimal point was found.

Thanks to its genetic algorithm methods, improved local search methods, and the distance and merit filters, the Evolutionary Solver performs exceedingly well on smooth global optimization problems, and on many non-smooth problems as well.

The local search methods range from relatively "cheap" to "expensive" in terms of the computing time expended in the local search step; they are listed roughly in order of the computational effort they require. On some problems, the extra computational effort will "pay off" in terms of improved solutions, but in other problems, you will be better off using the "cheap" Randomized Local Search method, thereby spending relatively more time on the "global search" carried out by the Evolutionary Solver's mutation and crossover operations.

In addition to the Local Search options, the Evolutionary Solver employs a set of methods, corresponding to the four local search methods, to transform infeasible solutions – generated through mutation and crossover – into feasible solutions in new regions of the search space. These methods, which also vary from "cheap" to "expensive," are selected dynamically (and automatically) via a set of heuristics. For problems in which a significant number of constraints are smooth nonlinear or even linear, these methods can be highly effective. Dealing with constraints is traditionally a weak point of genetic and evolutionary algorithms, but the hybrid Evolutionary Solver is unusually strong in its ability to deal with a combination of constraints and non-smooth functions.

Global Search

Name	globalSearch

Default	1 – Genetic Algorithm
Min	0 – Scatter Search
-Max	1 – Genetic Algorithm
Type	Integer

If this option is set to 1 (Genetic Algorithm), then the Evolutionary Solver will use methods from the literature on genetic algorithms (its traditional methods) to solve the model. Otherwise, the Evolutionary Solver will use methods from the literature on scatter search. On some models, the scatter search algorithm will result in better answers in less time when compared to the genetic algorithm. However, for other models, the genetic algorithm may be more successful. Since the scatter search algorithm tends to perform best, by a modest margin, on the majority of models, it is the default choice. But we suggest you try both algorithms with your model to see which works better for you.

Model Based Search

Name	modelBasedSearch
Default	0 – None
Min	0
Max	2
Type	Integer

0-None, 1- CPU Based, 2-GPU Based

This option takes effect only when the Legacy Mode option is set to False. When this option is set to "None", the new Scatter Search algorithm is used without any Model Based Search. When this option is set to either "CPU Based" or "GPU Based", an internal model of the problem is created (using the *Radial Basis Functions* method) which closely fits the original problem. The Evolutionary Engine uses this internal model to evaluate many points in parallel (either on the CPU or GPU **-** depending on the option setting) rather than calling the Interpreter to evaluate each of these points sequentially. Only the most promising of these points are sent to the Interpreter for actual evaluation using the new Scatter Search Algorithm. This new search method typically results in better solutions in less time when compared to using only the Scatter Search algorithm.

Evolutionary Parameters for Integer Problems

Where the other Solver enignes use the Branch & Bound method to solve problems with integer constraints, the Evolutionary Solver handles integer constraints on its own, and is subject to the limits set in the following parameters, Max Subproblems, Max Feasible Solutions, Tolerance, Max Time Without Improvement, and Solve Without Integer Constraints. Unlike the other Solver engines, the Evolutionary Solver *always* works on a series of subproblems, even if there are no integer constraints in the model – so these options are always important for the Evolutionary Solver.

Max Feasible Solutions

Name	maxFeasibleSols
Default	Unlimited*
Min	0
Max	Unlimited*
Type	Integer

The value for the Max Feasible Solutions parameter places a limit on the number of feasible solutions found by the evolutionary algorithm before the engine stops with the status result, *Branching and bounding node limit reached. Indicates an early exit of the algorithm*. A feasible solution is any solution that satisfies all of the constraints, including any integer constraints.

Max Subproblems

Name	maxSubproblems
Default	Unlimited*
Min	0
Max	Unlimited*
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the evolutionary algorithm before the engine stops with the Status result, *Branching and bounding maximum number of incumbent points reached. Indicates an early exit of the algorithm.* In the Evolutionary Solver, a subproblem consists of a possible mutation step, a crossover step, an optional local search in the vicinity of a newly discovered "best" point, and a selection step where a relatively "unfit" member of the population is eliminated.

Max Time without Improvement

Name	maxTimeNoImp
Default	30
Min	0
Max	Unlimited
Type	Integer

This option works in conjunction with the Tolerance option to limit the time the evolutionary algorithm spends without making any significant progress. If the relative (i.e. percentage) improvement in the best solution's "fitness" is less than the Tolerance parameter value, the engine stops with the Status result, *"No remedies" status. (All remedies failed to find better point.)* unless the evolutionary algorithm has discovered no feasible solutions at all, in which case the status is *Feasible solution could not be found*. If you believe that this stopping condition is being met prematurely, you can either make the Tolerance value smaller (or even zero), or increase the number of seconds allowed by the Max Time without Improvement option.

Solve Without Integer Constraints

Name	solveWithout
Default	0 **-** Off
Min	0
Max	1 – On
Type	Integer

If you solve your Problem with this parameter set to 1, the Solver *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem, which is often quite useful.

Tolerance

Name	intTolerance
Default	0
Min	0
Max	1.0
Type	Double

This option works in conjunction with the Max Time without Improvement option to limit the time the evolutionary algorithm spends without making any significant progress. If the relative (i.e. percentage) improvement in the best solution's "fitness" is less than the Tolerance parameter's value, the Evolutionary Solver stops as described below. Since the population may include members representing infeasible solutions, the "fitness" value is a combination of an objective function value and a penalty for infeasibility.

Mixed Integer Problem Parameters

This section describes parameters used when solving integer programming problems for all Solver engines *except* the LP/Quadratic Solver. The LP/Quadratic Solver's integer options are described in the section, *LP/Quadratic Solver MIP Parameters* above. This section explains all of the options that pertain to the solution of problems with integer constraints.

Integer Cutoff

Name	intCutoff
Default	2e+30
Min	-2e-30
Max	2e+30
Type	Double

This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a previous run of the same or a very similar problem – you can enter this objective value for the Integer Cutoff parameter. This allows the Branch & Bound process to *start* with an "incumbent" objective value (as discussed above under Integer Tolerance) and avoid the work of solving subproblems whose objective can be no better than this value. If you enter a value for this parameter, you must be *sure* that there is an integer solution with an objective value at least this good: A value that is too large (for maximization problems) or too small (for minimization) may cause the engine to skip solving the subproblem that would yield the optimal integer solution.

Integer Tolerance

Name	intTolerance
Default	0
Min	0
Max	1.0
Type	Integer

When you solve an integer programming problem, it often happens that the Branch & Bound method will find a good solution fairly quickly, but will require a great deal of computing time to find (or verify that it has found) the optimal integer solution. The Integer Tolerance setting may be used to tell the engine to stop if the best solution it has found so far is "close enough."

The Branch & Bound process starts by finding the optimal solution without considering the integer constraints (this is called the *relaxation* of the integer programming problem). The objective value of the relaxation forms the initial "best bound" on the objective of the optimal *integer* solution, which can be no better than this. During the optimization process, the Branch & Bound method finds "candidate" integer solutions, and it keeps the best solution so far as the "incumbent." By eliminating alternatives as its proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

```
Objective of incumbent - Objective of best bound
------------------------------------------------
              Objective of best bound
```

If the absolute value of this maximum percentage difference is equal to or less than the Integer Tolerance, the Solver will stop and report the current integer solution as the optimal result. If you set the Integer Tolerance to zero, the Solver will continue searching until all alternatives have been explored and the optimal integer solution has been found. This could take a great deal of computing time.

Max Feasible (Integer) Solutions

Name	maxIntegerSols maxFeasibleSols
Default	Unlimited*
Min	0
Max	Unlimited*
Type	Integer

The value for the Max Integer Solutions parameter places a limit on the number of "candidate" integer solutions found by the Branch & Bound algorithm before the engine stops with the status result, *Branching and bounding node limit reached. Indicates an early exit of the algorithm*. Each candidate integer solution satisfies all of the constraints, including the integer constraints; the engine retains the integer solution with the best objective value so far, called the "incumbent."

It is entirely possible that, in the process of exploring various subproblems with different bounds on the variables, the Branch & Bound algorithm may find the same integer solution (set of values for the decision variables) more than once; the Max Integer Solutions limit applies to the total number of integer solutions found, not the number of "distinct" integer solutions.

Max Subproblems

Name	maxSubproblems
Default	Unlimited*
Min	0
Max	Unlimited*
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the engine stops with the status result, *Branching and bounding maximum number of incumbent points reached. Indicates an early exit of the algorithm*. Each subproblem is a "regular" Solver problem with additional bounds on the variables. In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit.

Solve Without Integer Constraints

Name	solveWithout
Default	0 **-** Off
Min	0
Max	1 – On
Type	Integer

If you solve your problem with this parameter set to 1, the engine *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem.

Simulation Engine Settings

When "modelType": "simulation, the Monte Carlo Simulation engine is selected. The options below control simulation engine.

Simulation Random Seed

Name	randomSeed
Default	0
Min	0
Max	Unlimited*
Type	Integer

Setting the random number seed to a **nonzero value** (any number of your choice is OK) ensures that the *same* sequence of random numbers is used for each simulation. When the seed is zero, the random number generator is initialized from the system clock, so the sequence of random numbers will be different in each simulation. If you need the results from one simulation to another to be strictly comparable, you should set the seed.

Sampling Method

Name	**samplingMethod**
Default	1
Monte Carlo	1
Latin HyperCube	2
Sobol RQMC	3
Type	Integer

Use this option to select **Monte Carlo (1)**, **Latin Hypercube (2)**, or **Sobol RQMC** (3) sampling. In standard Monte Carlo sampling, numbers generated by the chosen random number generator are used directly to obtain sample values for the uncertain variables (PSI Distribution functions) in the model. With this method, the variance or estimation error in computed samples for uncertain functions is inversely proportional to the square root of the number of trials; hence to cut the error in half, four times as many trials are required.

RASON provides two other sampling methods than can significantly improve the 'coverage' of the sample space, and thus reduce the variance in computed samples for output functions. This means that you can achieve a given level of accuracy (low variance or error) with fewer trials.

- **Latin Hypercube Sampling.** Latin Hypercube sampling begins with a stratified sample in each dimension (one for each uncertain variable), which constrains the random numbers drawn to lie in a set of subintervals from 0 to 1. Then these one-dimensional samples are combined and randomly permuted so that they 'cover' a unit hypercube in a stratified manner. This often reduces the variance of uncertain functions.

- **Sobol numbers (Randomized QMC).** Sobol numbers are an example of so-called "Quasi Monte Carlo" or "low-discrepancy numbers," which are generated with a goal of coverage of the sample space rather than "randomness" and statistical independence. A "random shift" is added to Sobol numbers, which improves their statistical independence. Sobol numbers are frequently used in quantitative finance applications, where they are often effective at reducing variance.

Random Number Generator

Name	randomGenerator
Default	1
Park-Miller	0
CMRG	1
Well	2
Marsenne	3
HDR	4
Type	Integer

Use this option to select a random number generation algorithm. RASON includes an advanced set of random number generation capabilities – well beyond those found in other Monte Carlo products. In common applications, any good random number generator is sufficient – but for challenging applications (for example in financial engineering) that involve many uncertain variables and many thousands of trials, the advanced features of RASON can make a real difference.

Computer-generated numbers are never truly "random," since they are always computed by an algorithm – they are called *pseudorandom* numbers. A random number generator is designed to quickly generate sequences of numbers that are as close to statistically independent as possible. Eventually, an algorithm will generate the same number seen sometime earlier in the sequence, and at this point the sequence will begin to repeat. The *period* of the random number generator is the number of values it can generate before repeating.

A long period is desirable, but there is a tradeoff between the length of the period and the degree of statistical independence achieved within the period. Hence, RASON offers a choice of four random number generators:

- **Park-Miller (0)** "Minimal" Generator with Bayes-Durham shuffle and safeguards. This generator has a period of 2^{31}-2. Its properties are good, but the following choices are usually better.

- Combined Multiple Recursive Generator (**CMRG**) of L'Ecuyer **(1)** This generator has a period of 2^{191}, and excellent statistical independence of samples within the period.

- Well Equidistributed Long-period Linear (**WELL1024**) generator of Panneton, L'Ecuyer and Matsumoto. **(2)** This very new generator combines a long period of 2^{1024} with very good statistical independence.

- **Mersenne Twister (3)** generator of Matsumoto and Nishimura. This generator has the longest period of 2^{19937}-1, but the samples are not as "equidistributed" as for the WELL1024 and CMRG generators.

- **HDR (4)** Random Number Generator, designed by Doug Hubbard. Removes the requirement to distribute simulation trials and permits simulations running on various computer platforms to generate identical or independent streams of random numbers.

Random Number Streams

Name	randomStreams
Default	0

Single Stream	0
Double Stream	1
Type	Integer

You can use this option group to select a **Single Stream for all Uncertain Variables**, or an **Independent Stream for each Uncertain Variable**. Most Monte Carlo simulation tools generate a single sequence of random numbers, taking values consecutively from this sequence to obtain samples for each of the distributions in a model. This introduces a subtle dependence between the samples for all distributions in one trial. In many applications, the effect is too small to make a difference – but in some cases, found in financial engineering and other demanding applications, better results are obtained if independent random number sequences (streams) are used for each distribution in the model. RASON Decision Services offers this capability for Monte Carlo sampling and Latin Hypercube sampling; it does not apply to Sobol numbers.

If you use a **PsiSeed()** property function as an argument to a PSI Distribution function call, the uncertain variable defined by that distribution always has an independent stream of random numbers, regardless of the setting of this option.

CLT Threshold

Name	**cltThreshold**
Default	100
Min	1
Max	1000
Type	Integer

RASON Decision Services includes the ability to sum multiple independent random variables using compound distributions. A compound distribution generates values for the sum of N independent identically distributed uncertain variables. A distribution is made compound through the use of the PsiCompound() property.

When calculating a compound distribution, RASON Decision Services first tries to compute the distribution analytically. For example "=PsiExponential(par, PsiCompund(N))" can be computed as PsiGamma(N, par) If RASON is unable to compute a compound distribution analytically, but the frequency of the severity function (N) is greater than the value for the CLT Threshold option, then the distribution will be computed according to the Central Limit Theorem as PsiNormal(m, s). (The parameters m and s will be computed analytically from the corresponding analytical moments of the severity distribution.) Othewise, the compound distribution will be computed using Monte Carlo simulation to sum up N independent variates of the severity distribution. The maximum value allowed for this option is 1000 while the minimum value allowed is 1. The default setting is 100.

Censor Type

Name	censorType
Default	0
Min	0
Max	2
Type	Integer

Use this parameter only if you want to set global default bounds on the probability distributions of all uncertain variables. This option specifies the "units of measure" for the values you enter for the Lower Censor and Upper Cutoff options. Set this parameter to 0(None - the default) if you do not want to set these global bounds.

If you set this parameter to 1 (Percentile), then the Lower Censor and Upper Censor values must be between 0.01 and 0.99, and they specify percentiles of each uncertain variable's probability distribution. If you set this parameter to 2 (Std Deviation), then the Lower Censor and Upper Censor can be any positive or negative value, and they specify the number of standard deviations away from the mean for each uncertain variable.

When you use Censor bounds, random samples from the distribution that lie above the upper bound are set equal to the upper bound, and samples that lie below the lower bound are set equal to the lower bound; this causes a "buildup of probability mass" at the bounds – which is appropriate in some situations, but not in others.

Cutoff Type

Name	cutoffType
Default	0
Min	0
Max	2
Type	Integer

Use this parameter only if you want to set global default bounds on the probability distributions of all uncertain variables. This option specifies the "units of measure" for the values you enter for the Lower Cutoff and Upper Cutoff parameters. Set this parameter to 0(None - the default) if you do not want to set these global bounds.

If you set this parameter to 1 (Percentile), then the Lower Cutoff and Upper Cutoff values must be between 0.01 and 0.99, and they specify percentiles of each uncertain variable's probability distribution. If you set this parameter to 2 (Std Deviation), then the Lower Cutoff and Upper Cutoff can be any positive or negative value, and they specify the number of standard deviations away from the mean for each uncertain variable.

When you use Cutoff bounds, random samples from the distribution are effectively rescaled to lie within the lower and upper bounds.

Lower Censor

Name	lowerCensor
Default	-1e+30
Min	-1e+30
Max	1e+30
Type	Double

Use this parameter to set a lower "censor" bound for values sampled from the probability for a specific Uncertain Variable. A lower censor bound causes all random samples drawn from the distribution that are less than this value to be set equal to this value. This means that there is a "buildup of probability mass" at the lower bound. If you do not want this effect, use the Lower Cutoff option instead.

Lower Cutoff

Name	lowerCutoff
Default	-1e+30
Min	-1e+30

Max	1e+30
Type	Double

Use this parameter to set a lower cutoff for values sampled from the probability for a specific Uncertain Variable. A lower cutoff has the effect of re-scaling the distribution so that random samples drawn from the distribution will never be less than the value you enter here. Because the distribution is re-scaled, there is no "buildup of probability mass" at the lower cutoff, as there is for the Lower Censor option.

Upper Censor

Name	upperCensor
Default	1e+30
Min	-1e+30
Max	1e+30
Type	Double

Use this parameter to set an upper "censor" bound for values sampled from the probability for a specific Uncertain Variable. An upper censor bound causes all random samples drawn from the distribution that are greater than this value to be set equal to this value. This means that there is a "buildup of probability mass" at the upper bound. If you do not want this effect, use the Upper Cutoff option instead.

Upper Cutoff

Name	upperCutoff
Default	1e+30
Min	-1e+30
Max	1e+30
Type	Double

Use this parameter to set an upper cutoff for values sampled from the probability for a specific Uncertain Variable. An upper cutoff has the effect of re-scaling the distribution so that random samples drawn from the distribution will never be greater than the value you enter here. Because the distribution is re-scaled, there is no "buildup of probability mass" at the upper cutoff, as there is for the Upper Censor option.

Formulas

This optional section can be used to perform calculations on data arrays or constant values which will be used in a constraint, objective function or uncertain function definition. In the example below, `c4`, `c5`, `c6,c7` and `c8` calculate formulas based on three uncertain variables, `uncVar1`, `uncVar2`, and `uncVar3`.

```
formulas : {
   c4: { formula: "5 * uncVar2" },
   c5: { formula: "5 + 2.5* uncVar1" },
   c6: { formula: " 4.1 * uncVar3" },
   c7: { value: 100 + uncVar2 * 7 },
   c8: { formula: "100 + 80* uncVar1" }
   },
```

Note: The RASON modeling language supports all but a few of Excel's functions[5] which means that you can write a formula easily using functions such as SUM, SUMPRODUCT, etc. along with operators such as + and *. You can define arrays and use Excel functions that return vector and matrix results and access your data from within an Excel worksheet or a database.

See the table below for the properties available in the formula section of your RASON model.

Data Property	Type	Explanation
`aliasName`	`aliasName: “num_parts_inventory”`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver’s Deploy Model button, if a block of cells containing formulas is assigned a defined name in the Excel Solver model.
`name`	`name: "parts"`	Use this property to define the array name.

`dimensions`	`dimensions: [3,1]`
	`dimensions: [3]`
	`dimensions: [1,3]`
	`dimensions: [3,2]`
		Defines a 1 – dimensional vertical array with 3 elements.
		Defines a 1-dimensional vertical array with 3 elements.
		Defines a 2 – dimensional horizontal array with 3 elements.
		Defines a 2 – dimensional array with 3 rows and 2 columns.
		All arrays are 1 – based. If missing, array shape will be defined by the shape of the value property; however, for easier readability of the code, the use of the *dimensions* property is recommended.

`value`	`value: [1, 1, 1]`	Sets the value of the array. While it is unlikely that this property would be required within `formulas`, as typically the value of an object will be computed by `formula`, it is permissible. See the example model, RGSpace2.json for an example. If dimensions property is missing, the shape of the variable array will be determined by the shape of the value property. However, it is recommended that the *dimensions* property be used for readability purposes.

[5] Note: Excel functions not supported by the Rason modeling language are: Call(), Cell(), CubeX(), EuroConvert(), GetPivotData(), HyperLink(), Indirect(), Info(), Offset(), RegisterID(), PivotDim(), PivotCube(), FormulaText(), Dollar(), Fixed(), Replace(), Search(), Text() and SqlRequest().

.

formula	formula: "5 + 2.5*temp2" formula: "MATOP(Supply, 'min', '+', transpose(Demand))"	Enter a formula to calculate a result or array which will be used in a constraint, uncertain function or in the objective function.
comment	comment: "partsReq" array holds the number of parts required to produce each product"	Enter a comment here to describe the data.

Index Sets

RASON uses index sets exclusively to dimension tables and arrays. Typical mathematical programming models include multiple tables and arrays indexed over various index sets. An index set should be created at the beginning of the model to establish a basis of order for each dimension appearing in a table or array. Otherwise, the user will be required to keep track of and maintain the correct order of elements in all arrays and tables present in the model. An index set is always a 1-dimensional array and *must* be defined within the *indexSets* section of the RASON model.

The example code below creates two ordered sets, *part* and *prod*. The *part* set contains five items (in order as entered): chas, tube, cone, psup, and elec while the *prod* set contains 3 items: tv, stereo, and speakers. An *indexSet* is always defined as a JSON object{}.

```
indexSets: {
        part: {
            value: ['chas', 'tube', 'cone', 'psup', 'elec']
        },
        prod: {
            value: ['tv', 'stereo', 'speaker']
        }

    },
data: {
        parts: {
            indexCols: ['part', 'prod'],
            value: [

                ['chas', 'tv', 1],
                ['elec', 'stereo', 1],
                ['tube', 'tv', 1],
                ['cone', 'tv', 2],
                ['cone', 'stereo', 2],
                ['chas', 'stereo', 1],
                ['cone', 'speaker', 1],
                ['psup', 'tv', 1],
                ['psup', 'stereo', 1],
                ['elec', 'tv', 2],
                ['elec', 'speaker', 1]
            ]
        },
        profits: {
            dimensions: ['prod'],
            value: [75, 50, 35]
        },
```

In the *data* section, a table is created that uses the dimensions *part* and *prod* as the index columns. Since the index set *prod* exists, we can dimension the profits array according to this set in order to assign the correct profit values to the appropriate products.

Members in a set are recorded as strings regardless of the name given to the element. Typically, the members of the set describe or represent examples of the same type of item, i.e. product lines, cities, countries, inventory, etc. The names of these items may be familiar (TVs, Stereos, Speakers) or not (Part 567, Part 987, Part 123). Regardless, the names of the set objects must be surrounded by a single quote and the entire set must be surrounded by [], for example: *['TVs', 'Stereos', 'Speakers'].*

Note: The RASON modeling language does not consider upper and lower case letters distinct so one could not include TVs, tvs, and Tvs in the same set. The name of an indexed set may be made up of letters, numbers, or symbols.

A set may also be a collection of numbers. Numbers will be treated as strings when used in an index set. In the example below, the index set birth contains five elements: 1950, 1951, 1952, 1953, and 1954.

```
indexSets :{

          birth: { value: [1950, 1951, 1952, 1953, 1954] }

       }
```

Now let's assume that the following table is created that lists the year of birth and whether or not the person is still alive (1) or not (0).

```
data: {
      status: {
            indexCols: ['birth'],
            value: [
                 [1950, 0],
                 [1951, 1]
                 [1952, 1],
                 [1953, 0],
                 [1954, 1]
            ]
        },
```

To refer to the coefficient for the year 1952, we would use: `status['1952']`.

Note: If we were to use `status[1952]` (no quotes), this would obtain the 1,955th element of the *status* table, which does not exist in this particular example. The only place you must reference index set components as strings is within the index operator [ ] when attached to a table name.

This set contains five years as elements: 1950, 1951, 1952, 1953 and 1954. We could have specified this same set by using the upper and lower properties to specify the lower and upper bounds of the set, respectively.

```
indexSets :
       [
          { name: "birth", lower: 1950, upper: 1954
        ],
```

In the following example, the years are not given in chronological order.

```
       indexSets: {
  { name: "years", value: [1958, 1930, 1940 , 1954, 1925]},
}
```

In this case, the order of the set is the same as the order entered. .

However, in the example below, the *sort* property ensures that the *years* index is sorted alphabetically in ascending order. (Note: The properties *sort* and *sortIndexCols* perform the same function.)

```
indexSets: {
  { name: "years", value: [1958, 1930, 1940, 1954, 1925], sortIndexCols: true },
}
```

Note: In the current implementation of RASON, operations over index sets are not supported.

See the sections *Array Formulas* and/or *Tables*, appearing later on in this guide, for information on how to reference an index set in an array or table.

Model Description

Use modelDescription to add a text string containing the description of the model (optional).

```
{
  "modelName":  "CollegeFundModel2",
  "modelDescription":  "This is a simulation model that simulates activity
  in a college savings account.",
…
}
```

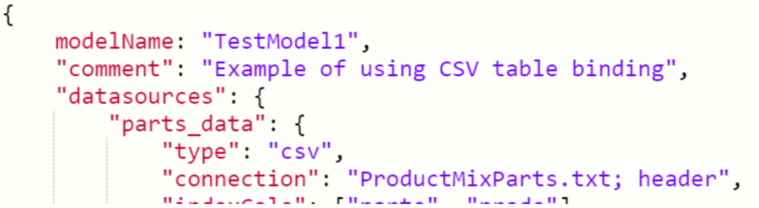
Model Name

Use modelName to assign a name to the RASON model, optional. RASON V2020 supports both named and unnamed models A model becomes "named" either by including modelName: "name" in the RASON model text and then calling POST rason.net/api/model or by simply calling POST rason.net/api/model/<name> or both. Either call returns a Location header with a new resource ID that identifies this unique model instance. The "name" must be unique among models with a user's account. An unnamed model, or a model not containing `modelName`, has no name.

- A model may be named by using a REST API call to POST rason.net/api/model/{name}, i.e. POST rason.net/api/model/TestModel2. See the example response below, note the resource ID (`2590+TestModel2+2019-11-22-17-53-20-176350`) that identifies the unique model instance.

  ```
  Location: https://rason.net/api/model/2590+TestModel2+2019-11-22-17-53-
  20-176350
  ```

- **A model may also be named by using the property `modelName`: `"name"` in the body of the RASON model. In the example code below, the model name is "TestModel1". The RASON model will be "named" with a call to POST rason.net/api/model or POST rason.net/api/model/{name}. If using POST rason.net/api/model/<name>, the name given to the `modelName` property and the name passed to POST rason.net/api/model/<name> *must* be identical, otherwise, an error will be returned.**



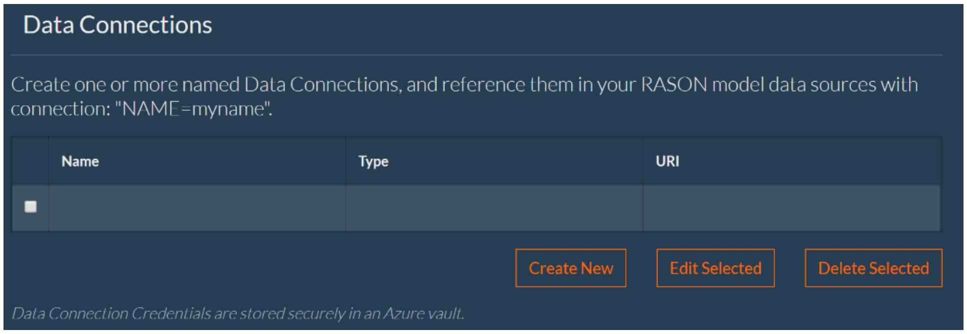
```
{
    modelName: "TestModel1",
    "comment": "Example of using CSV table binding",
    "datasources": {
        "parts_data": {
            "type": "csv",
            "connection": "ProductMixParts.txt; header",
            "indexCols": ["parts", "prods"]
```

See the example response below, again, note the resource ID (`2590+TestModel1+2019-11-22-18-25-40-015004`) that identifies the unique model instance.

```
Location: https://rason.net/api/model/2590+TestModel1+2019-11-22-18-25-40-
015004
```

The RASON Server maintains a simple, one-level directory of named models as ordinary text files using Azure file storage.

Additionally, a RASON model may be saved to the user's OneDrive for Business account. If the model is stored in OneDrive for Business, the user must give the RASON server permission to access the account on OneDrive by adding a Data Connection on the MyAccount page at www.RASON.com.



Data Connections

Create one or more named Data Connections, and reference them in your RASON model data sources with connection: "NAME=myname".

□	Name	Type	URI
□			

Data Connection Credentials are stored securely in an Azure vault.

For more information on creating a Data Connection, see the *Data Connections* section within the *RASON Subscriptions* chapter in the *RASON User Guide*. Note: There is a 4 MB limit on the size of files written back to OneDrive or OneDrive for Business.

Model Settings

In this optional section, you may specify options relevant to the solve such as the number of optimizations or simulations to run, the number of trials to perform in a simulation model, what Stochastic Transformation method to use when solving a stochastic optimization model, if a nonsmooth model should be transformed, etc.

The example code below sets Transform Nonsmooth to True and the number of optimizations to 2. The order in which the options appear is not relevant.

```
modelSettings: {

        transformNonSmooth: true, numOptimizations:2

    },
```

See below for a complete description of each available model option.

*Note: *Unlimited* in the tables below equals the maximum 32-bit integer setting, 2,147,483,647.

Active Sheet

Name	activeSheet
Used (only) during automatic conversion from Analytic Solver (desktop or online) to the RASON modeling language.	

Auto Adjust Chance Constraint

Name	chanceAutoAdjust
Default	False
Min	False
Max	True

Type	True or False

Set this option to True if you want your Robust Counterpart model to be automatically re-solved while adjusting the size of uncertainty sets created for chance constraints, in an effort to find a better (less conservative) solution. This can take significantly more time for a large model. If this option is set to False (the default), the Robust Counterpart model will not automatically re-solve the model.

Big M Value

Name	BigM
Default	1E+6
Min	-1.00E+30
Max	+1.00E+30
Type	Double

Use this option to set a "Big M" constant value to be used in newly generated constraints that result from a Nonsmooth Model Transformation. The default value is 1E6 or 1 million – but if you are using transformation features, you should ensure that this value is correct for your model: It *must* be bigger than any numeric value that may appear in your intermediate calculations (for example, bigger than any value a in an expression IF(a>=b,…)) but it *should not* be excessively large. If your value for the Big M option is *smaller* than the largest value that occurs in your intermediate calculations, the generated constraints will not have the desired effect, and your solution will **not be valid** for your original problem. If your Big M value is *too large*, the transformed model will be poorly scaled, and the Solver engine will likely encounter problems with numerical stability as it performs computations with your too-large values. So it pays to investigate the results computed by your what-if spreadsheet model, and set the Big M option appropriately.

Chance Constraint Use

Name	chanceConstraintNorm
Options	L1, L2, Linf, D

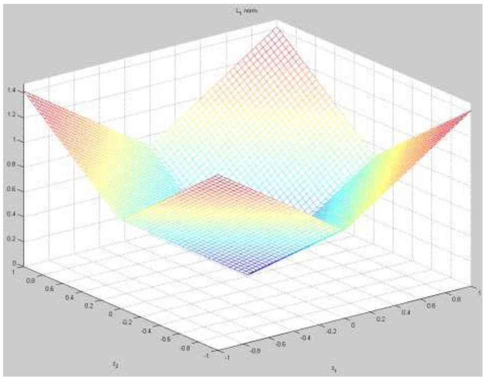
This option determines the norm (distance measure) used to constrain the size of uncertainty sets in the Robust Counterpart model. Select from the **L1 Norm**, **L2 Norm**, **L-Inf Norm**, or **D Norm** (the default). The D norm is equivalent to the intersection of the L1 norm and L-Inf (infinity) norm. If you choose the L2 norm, the Robust Counterpart model will be a SOCP (second order cone programming) model, which requires an SOCP or smooth nonlinear solver (such as the SOCP Barrier Solver or GRG Nonlinear Solver). If you choose the L1, L-Inf or D norm, the Robust Counterpart model will be an LP (linear programming) model that can be solved efficiently with an LP, QP, or SOCP Solver.

Uncertainty Sets and Norms

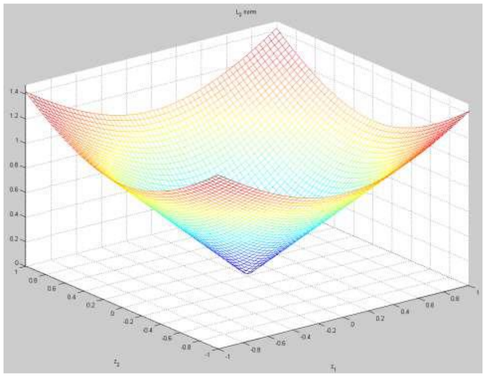
If a chance constraint is linear in the decision variables, you can use the *USet* (uncertainty set) criterion, in lieu of the VaR or CVaR criterion. The advantage of using this criterion is that the robust counterpart model more accurately reflects the degree to which you want the chance constraint to be satisfied, which can lead to less conservative solutions, with better objective values. Consider a constraint: $a_1x_1 + a_2x_2 + ... + a_nx_n \leq b$ where $=a_1x_1 + a_2x_2 + ... + a_nx_n$ is in A1, *b* is in B, and some or all of the coefficients a_i may depend on uncertain variables z_1, z_2, ... It is useful to think of the vector $[a_1\, a_2 ...\, a_n]$ as having a *nominal* or expected value, and a *variation* from this value for each realization of the uncertain variables. A constraint of the form **USet**$_\Omega$ **A1 <= B1** specifies that A1 <= B1 must be satisfied for ***all*** *variations* from the *nominal* value of $[a_1\, a_2 ...\, a_n]$ that do not exceed a bound Ω, measured by a norm. The *uncertainty set* includes all the points formed by adding a vector of allowed variations to the vector of nominal values; the bound Ω is often called the *budget of uncertainty* for the constraint.

RASON allows you to choose among four different norms to measure variation from the nominal value: The L1, L2, L Inf (Infinity) and D norms. (One choice of norm applies to all chance constraints.) The graphs shown on the following pages may help you visualize the shape of the uncertainty set (based on two uncertain variables z_1, z_2) for each of the norms. The D norm represents the intersection of the L1 norm and the L-Inf norm; thus it can define a 'tighter' uncertainty set than either of these norms alone. For the D norm, Ω can be interpreted as a bound on the *number* of coefficients $[a_1\, a_2 ... a_n]$ that depart from nominal values. When the D norm is used, the robust counterpart of a stochastic LP problem is a (larger, conventional) LP problem; when the L2 norm is used, the robust counterpart is an SOCP problem.

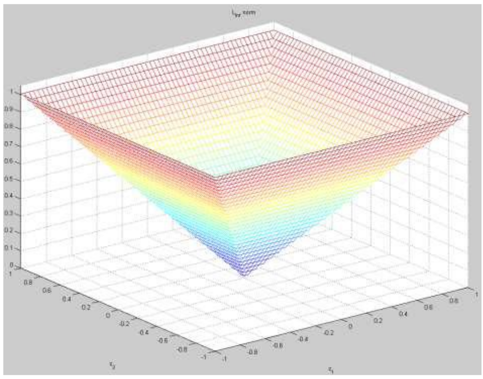
L1 Norm



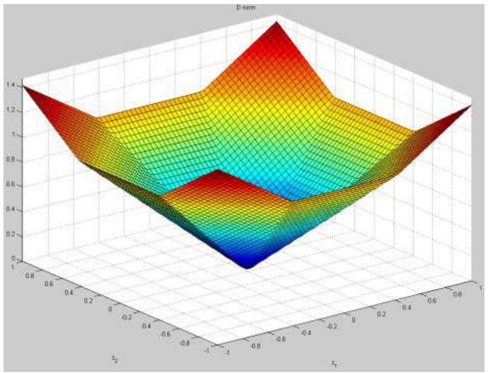
L2 Norm



L Inf Norm



D Norm



Nonsmooth Model Transformation

Name	transformNonsmooth
Default	False

Min	False
Max	True
Type	True or False

Use this option to choose whether Solver will attempt to transform constraints in your model that are *non-smooth* functions of the decision variables into equivalent linear constraints that depend on newly introduced binary integer and continuous decision variables. Your model will be automatically diagnosed and if it contains non-smooth functions that are candidates for transformation, Solver will attempt the transformation and will diagnose the resulting expanded model. If your model is successfully transformed, you should be sure to check, and probably adjust, the Big M Value option.

A simple example is the constraint:

```
constraint1: {
          formula: "If(test1=0,test2,test3)",
          upper: 100
         },
```

If `test1` is (or depends on) a decision variable, this constraint is *non-smooth* – in fact *discontinuous* – which means that the model cannot be solved to optimality by either linear programming (fastest and most reliable) or smooth nonlinear optimization.

Assuming for simplicity that test1 *is* a decision variable that is non-negative, this constraint can be transformed by introducing a new binary integer variable Binary1, and a new constraint test1 <= *BigM* * Binary1, where *BigM* is a constant larger than any possible value for Binary1 (you can set this value with the Big M Value option).

The IF function is replaced internally with =test3*Binary1+test2*(1-Binary1). Now when Binary1=0, the first term (test3 * Binary1) is forced to 0 (test3*0 = 0), and the function evaluates to test2 (test2 * (1-0)); when Binary1=1, the second term (test2 * (1 – Binary1) is forced to 0 (test2 * (1-1)), and the function evaluates to test3 (test3 * 1). The non-smooth IF function is transformed into a set of linear functions, so a faster and more reliable linear programming Solver can potentially be used – but the overall size of the model is increased. The Platform can perform much more complex transformations automatically, for constraints involving functions IF, AND, OR, NOT, MIN, MAX, and the relational operators <=, = and >=. Such transformations can result in a significantly larger model, but *if* the resulting model is entirely linear, this can be more than offset by the faster speed and reliability of a linear programming Solver.

Optimizations to Run

Name	numOptimizations
Default	1
Min	1
Max	Unlimited*
Type	Integer

Use this property to set the number of optimizations to run. This is useful only if you've defined one or more optimization parameters, *PsiOptParam()*. You can use these features to run multiple, parameterized optimizations. For example, in a product mix example you could define an optimization parameter to be varied from (say) $100 to $50 to reflect different selling prices. If you set the Optimizations to Run value to 6, the Solver engine would solve 6 portfolio optimization problems: the first one would use a selling price of $50, the second would have a selling price of $60, and so on through the 6th problem with a selling price of $100. The results of all 6 optimizations will be returned in the Result.

For more information on PsiOptParam(), please see the Parameters section explanation below.

Random Seed

Name	randomSeed
Default	1
Min	1
Max	Unlimited*
Type	Integer

Setting the random number seed to a **nonzero value** (any number of your choice is OK) ensures that the *same* sequence of random numbers is used for each simulation. When the seed is zero, the random number generator is initialized from the system clock, so the sequence of random numbers will be different in each simulation. If you need the results from one simulation to another to be strictly comparable, you should set the seed. To do this, click the spinner next to the **Random Seed** edit box, or type the number you want into the box.

You can specify a random seed for each uncertain variable if you wish (in the `uncertainVariables` section) by including the **PsiSeed()** property function as an argument to the PSI Distribution function formula for that variable. The seed value you set using the model option *randomSeed* affects only uncertain variables that do *not* have PsiSeed() property functions.

Run Specific Optimization

Name	optimizationIndex
Default	
Min	1
Max	NumOptimizations
Type	Integer

The specific optimization the Solver will perform, if multiple optimizations are defined. If PsiOptParam() exists, the parameter for the *optimizationIndex* specified will be used.

Run Specific Simulation

Name	simulationIndex
Default	1
Min	1
Max	NumSimulations
Type	Integer

The specific simulation that will be performed if multiple simulations are defined. If PsiSimParam() exists, the parameter for the *simulationIndex* specified will be used.

Simulation Optimization

Name	simulationOptimization
Default	False
Min	false
Max	True
Type	True or False

Use this option to determine how an optimization model with uncertainty will be solved. Your optimization includes uncertainty if the formula for the objective, or any constraint, depends (directly or indirectly) on an uncertain variable cell, where you've used a PSI distribution function (such as PsiNormal). If `simulationOptimization="True"`, then Simulation Optimization will be used to solve the RASON model. This is the most general method (it can handle nonlinear and non-smooth models), but is also the slowest and least reliable.

Simulations to Run

Name	numSimulations
Default	1
Min	1
Max	Unlimited*
Type	Integer

Use this property to set the number of simulations to run. This is useful only if you've defined one or more simulation parameters, *PsiSimParam()*. You can use this feature to run multiple, parameterized simulations. For example, in an airline yield management model where the number of "noshows" for a departing flight depends on the number of tickets sold, you could define a simulation variable for the number of tickets sold as a simulation parameter, varied from (say) 100 to 150. If you set the Simulations to Run value to 51, 51 simulations will be performed: the first simulation would use 100 tickets sold, the second would use 101 tickets sold, and so on through the 51st problem with 150 tickets sold. The results for all 51 simulations will be included in the Result. For more information on PsiSimParam() please see the Parameters section explanation below.

Use Sparse Variables

Name	Sparse
Default	0 (False)
Min	0 (False)
Max	1 (True)
Type	Boolean

Use this option to determine whether Rason should operate in (its own) Sparse mode or Dense mode. The default setting is False, meaning that the Interpreter operates in its Dense mode.

If you set this option to True, Rason will use its own Sparse mode, which can save memory when your optimization model is sparse, but possibly at the expense of extra time, since a Structure analysis is *always* performed when analyzing or solving.

Use Sparse Cubes

Name	sparseCubes
Default	0 (False)
Min	0 (False)
Max	1 (True)
Type	Boolean

Use this option to determine whether Rason should calculate a cube defined by PsiCube() or PsiTableCube() using Sparse mode or Dense mode.

Most large cubes are *sparse* in nature. While they may contain thousands of elements, in practice, not all combinations of dimension elements are possible.   Hence, not all will define a model function during the Psi Interpreter's evaluation of the problem. This means that most cubes will provoke output results as sparse cubes (with missing constraints). Such sparsity in a cube, also known as structural sparsity, can be exploited to save memory and gain speed.

A sparse cube is defined by missing *values in cells* for PsiCube() and by missing *records* for PsiTableCube(). If this option is equal to False and you have defined a cube using PsiCube() or PsiTableCube(), elements missing from the cube will be considered equal to 0. If you set this option to True, you have defined a cube using PsiCube() with missing values or PsiTableCube() with missing records, *and* the percentage of elements missing or empty is more than 30% of the total possible cube elements, those missing elements or records will not be included in the model.

For an example of how to use a sparse cube see the *Dimensional Modeling* chapter in the *Frontline Solvers User Guide*.

Stochastic Transformation

Name	transformStochastic
Options	deterministicEquivalent or robustCounterpart

Use this option to determine how an optimization model with uncertainty will be solved. Your optimization includes uncertainty if the formula for the objective, or any constraint, depends (directly or indirectly) on an uncertain variable cell, where you've used a PSI distribution function (such as PsiNormal). Stochastic Transformation works only with *linear* models that include uncertainty; it uses either stochastic programming or robust optimization methods to solve the problem (see the Transformation options for further information).

You can choose **Deterministic Equivalent** or **Robust Counterpart**. This transformation can succeed only if your objective and constraints are linear functions of the decision variables (they can also depend on uncertain variables).

Use this option to determine if an attempt is made to transform your optimization model *with* uncertainty into a conventional optimization model *without* uncertainty: either the Deterministic Equivalent model (as used in stochastic linear programming), or a Robust Counterpart model (as used in robust optimization).

The result of a successful transformation is a conventional linear programming model, but with considerably more decision variables and constraints than the original model. Generally, the Robust Counterpart model is much smaller than the Deterministic Equivalent model, but the solution of this model may be only an approximate (and conservative) solution of the original problem.

Trials Per Simulation

Name	numTrials
Default	1
Min	1
Max	100,000,000
Type	Integer

Use this property to set the number of Monte Carlo trials to run in each simulation. The default value is 1,000 trials, which is enough for a good statistical sample in most models. But applications such as estimating the value of options and other derivatives may need a higher number of trials.

Objective

This optional section is used for defining a normal, expected, or chance objective function in an optimization, stochastic optimization or simulation optimization model. In the example below the objective is being maximized. In return, the final value of the objective will appear in the result.

```
objective: {
        obj: {
            type: "maximize",
            formula: "sumproduct(parts, profits)",
            finalValue: []
        }
    }
```

Please see the table below for all input properties available in `objective`.

Input Property	Example	Definition
`aliasName`	`aliasName: “num_parts_inventory”`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver’s Deploy Model button, if the cell containing the objective function is assigned a defined name in the Excel Solver model.
`comment`	`comment: "Calculate Profit"`	Enter a comment here to describe the objective function. (Optional)
`name`	`name: "obj"`	Enter a name for the objective function. (Optional)

`type`	1. `type: "min"`
	`type: "minimize"`
	2. `type: "max"`
	`type: "maximize"`
		Required. Defines the problem as a “maximum” or “minimum” optimization model.

`formula`	`formula: "sumproduct(X,partsReq[1,])"`	Calculates the objective function; formula must resolve to a scalar. (Required.)

	where X and partsReq are predefined arrays.	

`chanceType`	`chanceType: "ExpVal"`
	`chanceType: "VaR"`
	`chanceType: "CVaR"`
	`chanceType: "USet"`
		Objective must contain uncertainties.
		Expected (ExpVal) – Sets the objective value to calculate the expected value of the objective or average over all simulation trials.
		Value at Risk (VaR) – Specifies that the chanceProbability percentile will be maximized.
		Conditional Value at Risk – Specifies the expected value of all the realizations of the objective up to the chanceProbability percentile will be maximized.
		Uncertainty Set – Specifies the objective must be maximized (or minimized) for all variations from the nominal value that does not exceed the chanceProbability.

`chanceProbability`	`chanceProbability: 0.95`	Defines the percentile for use with VaR, CVaR, and USet objective types.

An output property must be specified within the objective definition as an empty array.

Output Property	Example	Definition
`finalValue`	`finalValue: []`	Creates an empty array to hold the final objective value for the objective function.
`initialValue`	`initialValue: []`	Creates an empty array to hold the initial value for the objective function.

The Objective Function

The quantity you want to maximize or minimize is called the *objective function*. This could be a calculated value for projected profits (to be maximized), or costs, risk, or error values (to be minimized). If the objective function depends on uncertainties, we must specify how we want to 'optimize' this function. The most common practice is to maximize or minimize the *expected value* (informally, the mean value) of the objective, over all realizations of the uncertainties. Instead of maximizing or minimizing the *expected value* of a function of the decision variables and uncertainties, you can maximize or minimize a *measure of the uncertainty* in a function. The objective will be converted to the form max *t* or min *t*, where *t* is a new variable, and a *chance constraint* will be inserted (A1 >= *t* or A1 <= *t)* with the measure of uncertainty that you specify.

Implicit and Explicit Forms for the Objective

When you set `chanceType: "ExpVal",` your objective cell will be treated as implicitly containing **E**[*objective*], or using sample realizations of the uncertainty PsiMean(*objective*).

Parameters

The RASON Modeling language supports two kinds of parameters: optimization and simulation parameters.

- An **optimization parameter** (PsiOptParam) is automatically varied when you perform multiple optimizations. An optimization parameter is defined in the `parameters` section. The model option, `numOptimization`s, must be set to an integer value greater than 1 in `modelSettings`. For more information on how to set this option, please see the Model Settings section above.

- A **simulation parameter** (PsiSimParam) is automatically varied when you perform multiple simulations. A simulation parameter is defined in the `parameters` section. The model option, `numSimulations`, must be set to an integer value greater than 1 in `modelSettings`. For more information on how to set this option, please see the Model Settings section above.

The example below is for an optimization parameter. A simulation parameter could be setup in the same way using PsiSimParam() and following the same steps below.

```
parameters: {
      desired: {
            formula: "PsiOptParam(0.10, 0.6)", finalValue: []
      },
}
```

In this example, `formula` contains the `PsiOptParam` function with two arguments "0.10" and "0.60". The first argument ("0.10") is the lower limit for the optimization parameter and the second argument ("0.60") is the upper limit for the optimization parameter. If `numOptimizations` is set to 6 in `modelSettings`, desired will equal 0.10 in the first optimization, 0.20 in the second optimization, 0.30 in the third optimization and ending with 0.60 in the sixth and final optimization. The increment value is calculated as: (upper – lower)/(numoptimizations-1).

Alternatively, a list of *values* may be passed to `PsiOptParam()`.

```
parameters: {
      desired: {
            formula: "PsiOptParam({0.18,0.08,0.45,0.35,0.50,0.60})",
      finalValue: []
      },
}
```

PsiOptParam will use the values in order as they are given. In the example above, in the first optimization desired will equal 0.18, in the 2nd optimization 0.08, in the third, 0.45, and so on. If `numOptimizations` is set to a number that is greater than the number of values passed, PsiOptParam() will return to the beginning of the sequence. For example, if `numOptimizations` is set to 8, desired will equal 0.18 in the 7th optimization and 0.08 and in the 8th optimization.

Variables

The "variables" section is required in an optimization, simulation optimization or stochastic optimization model. Here you will define a variable or a block of variables, specify the variable "type" (integer, binary, semicontinuous, all different or conic), stipulate if the variable is to be a recourse variable (for use with stochastic optimization), identify lower or upper bounds for the variable(s), add a comment to describe the variables, and/or assign an initial value to the variable or variable block. In return you may ask for the variable's final value, dual value, dual upper value and/or dual lower value. In the example code below, three variables are defined (x[0], x[1], and x[2]). All are given an initial value and lower bound of 0. In return, the variable's final value (`finalValue: []`) will be returned in the result.

```
variables: {
```

```
          x: { dimensions: [3], value: 0, lower: 0, finalValue: []
          }
     },
```

We also could have created the variable x array by using an alternate syntax, shown below.

```
variables : [
     { name: "X", value: [1.0, 1.0, 1.0], finalValue:[], dualUpper: [],
       dualLower: [], dualValue: [] }
     ],
```

Please see the table below for all input properties available in `variables`.

Input Property	**Example**	**Definition**
`aliasName`	`aliasName: “variables”`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver’s Deploy Model button, if a block of cells containing decision or recourse variables is assigned a defined name in the Excel Solver model.
`comment`	`comment: "number of parts to produce"`	Enter a comment here to describe the variable or block of variables. (Optional)
`name`	`name: "numProducts"`	Enter a name for a variable or block of variables. (Optional)
`dimensions`	1. `dimensions: [3,1]` 2. `dimensions: [3]` 3. `dimensions: [1,3]` 4. `dimensions: [3,2]`	1. Defines a 1 – dimensional vertical array with 3 elements. 2. Defines a 1 **-** dimensional vertical array with 3 elements. 3. Defines a 2 – dimensional horizontal array with 3 elements. 4. Defines a 2 – dimensional array with 3 rows and 2 columns. All arrays are 1 – based. If missing, variable array shape will be implicitly defined by the shape of the *lower*, *upper*, or *value* properties, however, for readability of the code, the use of the *dimensions* property is recommended.
`value`	`value: [1, 1, 1]`	Sets the initial values of the variables to "1". If dimensions property is missing, the shape of the variable array will be determined by the shape of the value property. If value is missing, the shape of the array will be determined by the lower or upper properties. However, it is recommended that the *dimensions* property be used for readability purposes.

`type`	1. `type: "int"`
	`type: "integer"`
	2. `type: "bin"`
	`type: "binary"`
		1. Defines the variable or variable block as **integers**.
		2. Defines the variable or variable block as **binary** integers.
		3. If present or missing, defines variable or variable block as a "real" variable.

	`3.    type:  "real"`
	`4.    type: "dif"`
	`      type: "alldif"`
	`5.    type: "sem"`
	`      type:  "semCon"`
		4. Defines the variable or variable block as having to be **alldifferent**. Variables in an alldifferent group always have a lower bound of 1 and an upper bound of N, where N is the number of variables in the group.
		5. Defines the variable or variable block as **semicontinuous** variables. This specifies that, at the solution, the variable must be either 0, or else a continuous value within a range, determined by the bounds on the variable

`recourse`	`recourse:  true`
	`recourse:  false`
		Defines the variable or variable block as recourse variable(s). The default setting for this option is false. For more information on these types of variables, please see the topics below. (Optional)

`lower*`	`lower: 0`
	`lower: [1, 2, 3]`
	`lower: ‘availInvent’ where availInvent is an array of constants.`
		Specifies the lower bound of the variable or variable block. If an array is passed and dimensions or value properties are missing, the shape of the variable array will be determined by the shape of the lower property. However, it is recommended that the *dimensions* property be used for readability purposes. If missing, the lower bound is defined as "unbounded". (Optional)
		Note: Only constant values are supported for this property. If a formula is provided to lower:[], the error: "Can not be parsed" will be returned.

`upper*`	`upper: 0`
	`upper: [1, 2, 3]`
	`upper: ‘availInvent’ where availInvent is an array of constants.`
		Specifies the upper bound of the variable or variable block. If an array is passed and dimensions or value properties are missing, the shape of the variable array will be determined by the shape of the upper property. However, it is recommended that the *dimensions* property be used for readability purposes. If missing, the upper bound is defined as "unbounded". (Optional)
		Note: Only constant values are supported for this property. If a formula is provided to upper:[], the error: "Can not be parsed" will be returned.

*The RASON Server currently ONLY supports constant values (i.e. 3, 8.54, etc.) or an array containing constant values for the *lower* and *upper* properties.

An output property must be specified within the variable definition as an empty array.

Output Property	**Example**	**Definition**
`dualLower`	`dualLower: []`	Creates an empty array to hold the Allowable Decrease for the variable or variable block. See the topic, *Interpreting Reduced Costs* below for more information on this property.
`dualUpper`	`dualUpper: []`	Creates an empty array to hold the Allowable Increase for the variable or variable block. See the topic, *Interpreting Reduced Costs* below for more information on each of these properties.
`dualValue`	`dualValue: []`	Creates an empty array to hold the reduced cost for the variable or variable block. The reduced cost for a variable is nonzero only when the variable is equal to its lower or upper

		bound. See the topic, *Interpreting Reduced Costs* below for more information on each of these properties.
`finalValue`	`finalValue: []`	Creates an empty array to hold the final value for the variable or variable block.
`initialValue`	`initialValue: []`	Creates an empty array to hold the initial value of the variable.
`indexValue`	`indexValue: []`	Creates an empty array to hold the index value for each variable in the block of variables.

Interpreting Reduced Costs

Reduced Costs are the most basic form of sensitivity analysis information. The reduced cost for a variable is nonzero only when the variable's value is equal to its upper or lower bound at the optimal solution. This is called a *nonbasic* variable, and its value was driven to the bound during the optimization process. Moving the variable's value away from the bound (or tightening the bound) will *worsen* the objective function's value; conversely, "loosening" the bound will *improve* the objective. The reduced cost measures the increase in the objective function's value *per unit increase* in the variable's value. The properties `dualLower` and `dualUpper` report the amount by which the variable's coefficient could be decreased or increased, respectively, without changing the dual value.

Recourse Variables

Conventional optimization deals with only one type of decision variable, which represents a decision that must be made 'here and now,' irrespective of any uncertainty in the model. We call this a normal or first-stage variable. If we are dealing with uncertainty that will be resolved in the future, then at some point the *array* of sample values for the uncertainty is effectively replaced by a *single* value, the *realization* of the uncertainty as it actually occurs.

If the situation we are modeling allows us to make certain decisions *after* the uncertainty becomes known, on a 'wait and see' basis, we can model these decisions with recourse variables, also called second-stage variables. (At the 'second stage,' the uncertainty has become known.)

Uncertain Functions

This section is required in a simulation model. In return you may ask for the final value and/or any of the nine statistical values computed for the uncertain functions such as mean, standard deviation, variance, skewness, etc. In the example code below, the uncertain function `revenue` is defined according to the formula property. In return, all 100 percentile values as well as the standard deviation will appear in the Result.

```
uncertainFunctions: {
        revenue: {
            formula: "price*(sold - refund_no_shows*Round(no_shows, 0) –
               refund_overbook*overbook)",
            percentiles: [],
            stdev:[]
        }
    }
```

We also could have created the uncertain function `revenue` by using an alternate syntax, shown below.

```
uncertainFunctions : {
    [ name: "revenue",
      formula: "price*(sold - refund_no_shows*Round(no_shows, 0) -
            refund_overbook*overbook)",
      percentiles: [],
      stdev: []
    ]
}
```

Please see the table below for all input properties available in `uncertainFunctions`.

Input Property	Example	Definition
`aliasName`	`aliasName: "OutputFunctions"`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver's Deploy Model button, if a block of cells containing uncertain functions are assigned a defined name in the Excel Solver model.
`comment`	`comment: "number of no-shows for flight"`	Enter a comment here to describe the uncertain function. (Optional)
`name`	`name: "no_shows"`	Enter a name for the uncertain function. (Optional)
`dimensions`	1. `dimensions: [3,1]` 2. `dimensions: [3]` 3. `dimensions: [1,3]` 4. `dimensions: [3,2]`	1. Defines a 1 – dimensional vertical array with 3 rows and 1 column. 2. Defines a 1-dimensional vertical array with 3 rows and 1 column. 3. Defines a 2 – dimensional horizontal array with 1 row and 3 columns. 4. Defines a 2 – dimensional array with 3 rows and 2 columns. All arrays are 1 – based. If creating a block of uncertain functions, you must use the dimensions property to define the size and shape of the array.
`formula`	`formula: "price*(sold - refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)"`	Use this property to calculate the uncertain function.

Statistics Functions

An output property must be specified within the uncertain function definition as an empty array. An output property for an uncertain function is a statistic function such as mean or standard deviation. During a simulation of 1,000 trials, 1,000 random sample values will be drawn for the uncertain function, and used to compute the statistic . Hence, you can think of the uncertain function as 'containing' an array of 1,000 values. But the output property (i.e. mean) will contain *one* value, which is the average or mean of the 1,000 values computed for the uncertain function.

Accessing Statistics from Different Simulations

Each PSI Statistic function is assigned to an optional *simulation* index (default=1). If you want statistic values from a specific simulation, say simulation #2, then you must set `simulationIndex: 2` in `modelSettings`. For more information, please see this option description in the *Model Settings* topic above.

A Note to Analytic Solver Users

- Selecting a specific simulation value within a statistic function is not supported. If interested on a particular simulation then users should use simulationIndex in modelSettings. Then output pertaining (only) to that simulation will be available in the results.
- PsiTheo functions are currently not supported in RASON.
- Statistics in an Analytic Solver model may be translated into a Rason model using Analytic Solver's Create App functionality when function arguments are explicitly provided as constants or cell references, if the cell references point to constant values. For example, =PsiPercentile(cell, **A1**) where A1 = 0.5 or simply =PsiPercentile(cell, 0.5).

Output Property	Example	Definition
absDev	absDev:[]	AbsDev returns the average of the absolute deviations of the specified uncertain function's sample values from their mean. This is also known as Mean Absolute Deviation (MAD), especially in time series analysis applications. It is defined as: $$MAD(X) = \frac{1}{n}\sum_{i=1}^{n}\lvert\mu - x_i\rvert$$ Here μ is the mean of the sample values.
bvar	`bvar(confidence_level): []`	BVaR returns the Value at Risk for the specified uncertain function *cell* at the specified 'confidence level,' which is better described as a *percentile* – for example, 0.95 or 0.99. (The "B" stands for "Basel" or "building block," and is used to distinguish this function name from a function named PsiVar().) In finance applications, the Value at Risk is the maximum loss that can occur at a given confidence level. In a distribution of returns or profits, losses would lie at the "left end" of the distribution and would be represented by negative numbers and smaller percentiles (say 0.01 or 0.05). But it is customary in Value at Risk analysis to treat losses as positive numbers at the "right end" of the distribution. Consider a Normal distribution, bvar (*percentile*) would be converted as –percentile(1- *percentile*).
citrials	citrials(confidence_level, tolerance):[]	CITrials returns the estimated number of simulation trials needed to ensure that the specified uncertain function's sample mean value (returned by the PsiMean() function) lies within the confidence interval specified by *confidence level* (for example 0.95 or 0.99) and the half-interval size given by *tolerance*. Note that this number of trials is sufficient only to ensure that the single output value specified by *cell* lies within the confidence interval. To ensure that *N* output cells lie within confidence intervals at level 1 - α (e.g. 0.95 = 1 – 0.05), use a confidence level of α / *N*.
coeffVar	coeffVar: []	CoeffVar finds the coefficient of variation for the specified uncertain function. This function is defined as the ratio of the standard deviation to the mean and is calculated as: $$C_v = \frac{\sigma}{\mu}$$ This statistic measures the magnitude of the variability in relation to the mean of the population.

correlation	correlation(unc_func_or_var): []
	Example where correlation statistic is used to return a correlation coefficient between two uncertain functions, uncFunc1 and uncFunc2.
	`uncertainFunctions: {`
	`  "uncFunc1": {"formula": "e17"},`
	`  "uncFunc2": {"formula": "d17",`
	`     "correlation(uncFunc1)": []}`
		The correlation statistic returns the Pearson product moment correlation coefficient betweentwo uncertain variables or functions. Correlation is a measure of linear dependence between two uncertain variables or functions. The correlation coefficient can take on values between -1 and +1. A correlation of -1 indicates a perfect negative correlation (the cells move linearly in opposite directions); a correlation of +1 indicates a perfect positive correlation (the cells move linearly in the same direction). If the two random variables are independent, then their correlation coefficient is zero; but if the correlation coefficient is zero, this does not necessarily mean that the two variables are independent.
		Pearson's product moment correlation coefficient between random variables *X* and *Y* is defined as:

		$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - E[X])(Y - E[Y])]}{\sigma_X \sigma_Y}$ In Monte Carlo simulation, this value is computed from the sample values *x*[] and *y*[] over *n* trials as: $r_{x,y} = \frac{n \sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{\sqrt{[n \sum_{i=1}^{n} {x_i}^2 - (\sum_{i=1}^{n} x_i)^2][n \sum_{i=1}^{n} {y_i}^2 - (\sum_{i=1}^{n} y_i)^2]}}$
count	count(type): []	The count statistic returns the number of trials of the specified *type* executed in the most recent simulation for the uncertain function. The *type* argument may be 0 for all trials, 1 for normal or 'success' trials (where the calculated uncertain function resulted in a number), or 2 for error or 'failed' trials (where the calculated uncertain function resulted in an error value).
cvar	cvar(percentile):[]	CVaR returns the conditional Value at Risk for the specified uncertain function *cell*, at the specified 'confidence level' which is better described as a *percentile* – for example, 0.95 or 0.99. The conditional Value at Risk is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs. Like PsiBVaR, PsiCVaR returns a loss as a positive number. It is computed as the negative of the mean value of the specified uncertain function for the trials that lie between Min:[] and Percentile(1-*percentile*), inclusive.
expGain	expGain:[]	PsiExpGain returns the average of all positive data multiplied by 1 - percentrank of 0 among all data. It is always a positive number.
expGainRatio	expGainRatio: []	PsiExpGain returns the expected gain ratio for a specified uncertain function. This function is calculated as: $ExpGaiRatio = \frac{ExpGain}{ExpGain + \lvert ExpLoss \rvert}$ This value ranges between 0 and 1 inclusive.
expLoss	expLoss:[]	ExpLoss returns the average of all negative data multiplied by the percentrank of 0 among all data. It is always a negative number.
expLossRatio	expLossRatio:[]	ExpLoss returns the expected loss ratio for a specified uncertain function. This function is calculated as: This value ranges between 0 and 1 inclusive.
expValMargin	expValMargin:[]	ExpValMargin is calculated as: *ExpValMargin = ExpGainRatio – ExpLossRatio.* This statistic ranges between -1 and 1 inclusive.
frequency	frequency(frequency_type, bin bounds): [] example: frequency(0, -125000, -100000 100000,200000): []	Frequency returns an array of frequencies describing the distribution of trial values for the specified uncertain function. Use this statistic to obtain the data required to draw a histogram chart in your application. The *freq_type* argument affects the contents of each element of the array result and can be 0, 1 or 2: 0 – Each element contains the frequency of trial values falling into the corresponding bin (like a probability density function)

		1 – Each element contains the cumulative frequency of trial values falling into the corresponding bin plus all lower bins (like a cumulative distribution function) 2– Each element contains the cumulative frequency of trial values falling into the corresponding bin plus all higher bins (like a reverse cumulative distribution function) In *RASON*, the *bin_bounds* argument is an array of values (e.g. { 5, 10, 15, 20 }) given in strictly increasing order. The number of elements in the array result will be one more than the number of values or cells in the *bin bounds* argument; the last element contains the number of trial values larger than the highest bin bound value.
indexValue	indexValue: []	Creates an empty array to hold the indexValue of the uncertain function.

kendallTau	kendallTau(unc_func_or_var): []
	Example where KendallTau statistic is used to return a correlation coefficient between two uncertain functions, uncFunc1 and uncFunc2.
	`uncertainFunctions: {`
	`  "uncFunc1": {"formula": "e17"},`
	`  "uncFunc2": {"formula": "d17",`
	`     "kendallTau(uncFunc1)": []}`
		The KendallTau statistic returns a non-parametric correlation coefficient (based on the relative ordering of ranks) between two uncertain variables or functions. This statistic can be used to determine how (if at all) the two uncertain variables or functions are correlated.
		The Kendall Tau rank correlation coefficient measures the ordinal association between two uncertain variables or functions. It is a measure of rank correlation. This correlation coefficient is high when observations have a similar rank between the two variables, and low when observations have a dissimilar rank between the two variables.

kurtosis kurt	kurtosis: [] kurt: []	Kurtosis returns the kurtosis for the specified uncertain function *cell*. Kurtosis is the 4th moment and measures the peakedness of the distribution of trial values. It is computed as: $kurtosis(X)=\frac{n\sum_{i=1}^{n}(x_i-\mu)^4}{\left[\sum_{i=1}^{n}(x_i-\mu)^2\right]^2}$ where μ is the mean of the trial values. A higher kurtosis indicates a distribution with a sharper peak and heavier tails, and that more of the variability is due to a small number of extreme outliers or values; a lower kurtosis indicates a distribution with a rounded peak and that more of the variability is due to many modest-sized values.
maximum max	maximum: [] max: []	Max returns the maximum value attained by the specified uncertain function over all the trials in the simulation.
mean average expectation	mean: [] average: [] expectation: []	Mean returns the mean value for the specified uncertain function. The mean or average value is the 1st moment of the distribution of trials and is computed as: $\mu=\frac{\sum_{i=1}^{n}x_i}{n}$

		The mean is frequently used as a measure of central tendency or the “middle” of an uncertain variable; but for skewed distributions, care must be taken in using the mean as a measure of central tendency because the mean is easily distorted by extreme outlier values.
meanCI	meanCI(confidence_level)	MeanCI returns the confidence “half-interval” for the estimated mean value (returned by mean: []) for the specified uncertain function, at the specified *confidence level* (for example 0.95 or 0.99). If μ is the value returned by mean:[] and δ is the value returned by meanCI(confidence_level):[], the true mean is estimated to lie within the interval $\mu - \delta$ to $\mu + \delta$. The *confidence level* can be interpreted as follows: If we compute a large number of independent estimates of confidence intervals on the true mean of the uncertain function, each based on *n* observations with *n* sufficiently large, then the proportion of these confidence intervals that contain the true mean of the function should equal the *confidence level*. If σ^2 (*n*) is the sample variance from *n* trial values, α = 1 – *confidence level*, and $t_{n-1,\ 1-\alpha/2}$ is the upper 1-α/2 critical point of the Student’s t-distribution with *n*-1 degrees of freedom, the confidence half-interval δ is computed as: $t_{n-1,1-\alpha/2}\sqrt{\frac{\sigma^2(n)}{n}}$ The confidence interval measures the precision with which we have estimated the true mean. Larger half widths imply that there is a lot of variability in our estimates. The above formula for the half-width assumes that the individual x_is are normally distributed; when this is not the case, the above formula still gives us an *approximate* confidence interval on the true mean of the uncertain function.
meanCIB	meanCIB(confidence_level, [lowerbound]):[]	MeanCIB returns the lower or upper bound of the confidence interval (half width) of the mean value for the specified uncertain function. *Confidence_level* – Enter the desired confidence level, i.e. 0.95 or 0.99. *lowerbound* – (Optional) Enter true for the lower (default) or false for the upper bound. *Example*: meanCIB(.99, true, 5) returns the lower bound for the 99% confidence interval for the distribution for simulation index 5.
median	median: []	Mean:[] returns the median value for the specified uncertain function. The median value is the 50th percentile of the distribution of trials and is computed as: $X_{(n+1)/2}$ if n is odd $\frac{X_{n/2} + X_{(n/2)+1}}{2}$ if n is even

		The median is a very useful statistic for measuring the center of a distribution.
minimum min	minimum: [] min: []	Min returns the minimum value attained by the specified uncertain function over all the trials in the simulation.
mode	mode: []	Mode returns the mode of the specified uncertain function. For discrete distributions, this is the most frequently occurring value (where the probability mass function has its greatest value). For continuous distributions, Mode() returns the *half-sample mode* as defined by D.R. Bickel, a robust estimator that is less sensitive to outliers than most other estimators of location.
percentileCI	percentile(percentile, confidence_level): []	PercentileCI returns the confidence "half-interval" for a given percentile (.01-.99) value for the specified uncertain function. This function is computed as: Lower: Percentile-PercentileCI, Upper: Percentile + PercentileCI. Since the output of PercentileCI is symmetric, the mean and median are *theoretically* the same, i.e. MeanCI(0.95) is expected to be approximately equal to PercentileCI(0.5, 0.95). This function together with MeanCI, MeanCIB, StdDevCI, CITrials and the newly added TargetCI, make up the confidence interval functions in RASON. Example: PercentileCI (0.95, 0.99,2):[] - Finds the confidence half-interval for the uncertain function using the 95th percentile and a confidence level of 99% for the 2nd simulation.
percentileD	percentileD(percentile): []	PercentileD returns a descending *percentile* (.01-.99) value for the specified uncertain function: This means that *m* (or *m*%) of the simulation trials have values less than the returned value, where *m* is the percentile.
percentiles	percentiles: [] percentile(confidence_level):[]	percentiles: [] returns all *percentile* (.01-.99) values for the specified uncertain function: This means that *m* (or 100*m*%) of the simulation trials have values less than the returned value, where *m* is the percentile. percentile(0.X) - Returns the specific percentile value. Values must be between 0.01 and 0.99.
range	range: []	Range returns the range of the specified uncertain function *cell*. The range is the difference between the maximum and minimum values attained in the distribution of trial values.
semiDev	semiDev(q, [target]): []	SemiDev returns the semideviation for the specified uncertain function, relative to the *target* if specified. If the *target* is omitted, the mean value is used. This is a *one-sided* measure of dispersion of values of the uncertain function. The semideviation is the square root of the semivariance, described directly below. If a *q* argument different from 2 is specified, SemiDev(): [] returns the *q*th root of the lower partial moment at power *q* of the uncertain function.
semiDev2	semiDev2([lowerdata]): []	SemiDev2 returns the standard deviation of the values in the distribution below or above the mean or the square root of SemiVar2.

		lowerdata – (Optional) Enter true for the lower (default) or false for the upper data. *Example*: semiDev2(true, 5) returns the standard deviation of the values below the mean for the distribution for simulation index 5.
semiVar	semiVar(q, target): []	SemiVar returns the semivariance for the specified uncertain function, if the argument *q* is omitted, or the 'lower partial moment' for the function, if an argument *q* different from 2 is specified. The semivariance is computed relative to the *target* if specified, or relative to the mean value if *target* is omitted. This is a measure of the dispersion of values of an uncertain function, but unlike the variance which measures (or penalizes) both positive and negative deviations from the target, the semivariance or lower partial moment is only concerned with *one-sided* deviations from the target. It is usually used in finance and insurance applications, when we are only concerned with downside risks (or loss in portfolio value). The semivariance is computed by summing only the downside differences from the target of all the trials, raised to the given power *q*, divided by the number of trials: $\frac{1}{n}\sum_{i=1}^{n}\left(t-x_i\right)_+^{q}$ $\left(x\right)_+ = \max\left(x,0\right)$ $t = \text{target value}$ All trials – not just the trials with downside deviations – are included in *n*. Again if *q* is different from 2, the result is called the 'lower partial moment.
semiVar2	sermiVar2([lowerdata]): []	SemiVar2 returns the variance of the values in the distribution below or above the mean. *lowerdata* – (Optional) Enter true for the lower (default) or false for the upper data. *Example*: =SemiVar2(true, 5) returns the variance of the values below the mean for the distribution for simulation index 5.
sigmaCP	sigmaCP(lower_limit, upper_limit): []	A Six Sigma index, PsiSigmaCP predicts what the process is capable of producing if the process mean is centered between the lower and upper limits. This index assumes the process output is normally distributed. $Cp = \frac{UpperSpecificationLimit - LowerSpecificationLimit}{6\hat{\sigma}}$ where $\hat{\sigma}$ is the estimated standard deviation of the process.
sigmaCPK	sigmaCPK(lower_limit, upper_limit,): []	A Six Sigma index, PsiSigmaCPK predicts what the process is capable of producing if the process mean is not centered between the lower and upper limits. This index assumes the process output is normally distributed and will be negative if the process mean falls outside of the lower and upper specification limits. $Cpk = \frac{MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

sigmaCPKLower	sigmaCPKLower(lower_limit, [simulation]): []	A Six Sigma index, PsiSigmaCPKLower calculates the one-sided Process Capability Index based on the lower specification limit. This index assumes the process output is normally distributed. $Cp, lower = \frac{\hat{\mu} - LowerSpecificationLimit}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaCPKUpper	sigmaCPKUpper(lower_limit): []	A Six Sigma index, PsiSigmaCPKUpper calculates the one-sided Process Capability Index based on the upper specification limit. This index assumes the process output is normally distributed. $Cp, upper = \frac{UpperSpecificationLimit - \hat{\mu}}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaCPM	sigmaCPM(lower_limit, upper_limit: []	A Six Sigma index, PsiSigmaCPM calculates the capability of the process around a target value. This index is referred to as the Taguchi Capability Index. This index assumes the process output is normally distributed and is always positive. $Cpm = \frac{\hat{C}p}{\sqrt{1+(\frac{\hat{\mu}-T}{\hat{\sigma}})^2}}$ where $\hat{C}p$ is the process capability (PsiSigmaCP), $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and T is the target process mean.
sigmaDefectPPM	sigmaDefectPPM(lower_limit, higher_limit)	A Six Sigma index, PsiSigmaDefectPPM calculates the Defective Parts per Million. $DPMO = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}}) +$ $1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}})) * 1000000$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaDefectShiftPPM	sigmaDefectShiftPPM(lower_limit, upper_limit, shift): []	A Six Sigma index, PsiSigmaDefectShiftPPM calculates the Defective Parts per Million with an added shift. $DPMOShift = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) +$ $1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)) * 1000000$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaDefectShiftPPMLower	sigmaDefectShiftPPMLower(lower_limit, shift)	A Six Sigma index, PsiSigmaDefectShiftPPMLower calculates the Defective Parts per Million, with a shift, below the lower specification limit.

		$DPMOshift, lower = (\delta^{-1}(\frac{LowerSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$ where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaDefectShiftPPMUpper	sigmaDefectShiftPPMUpper(upper_limit)	A Six Sigma index, PsiSigmaDefectShiftPPMUpper calculates the Defective Parts per Million, with a shift, above the lower specification limit. $DPMOshift, upper = (\delta^{-1}(\frac{UpperSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$ where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaK	sigmaK(lower_limit, upper_limit): []	A Six Sigma index, PsiSigmaK calculates the Measure of Process Center and is defined as: $1 - \frac{2*MIN(UpperSpecificationLimit-\hat{\mu},\hat{\mu}-LowerSpecificationLimit)}{UpperSpecificationLimit-LowerSpecificationLimit}$ where$\hat{\mu}$ is the process mean.
sigmaLowerBound	sigmaLowerBound(number_stdev)	A Six Sigma index, PsiSigmaLowerBound calculates the Lower Bound as a specific number of standard deviations below the mean and is defined as: $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaProbDefectShift	sigmaProbDefectShift(lower_limit, upper_limit, shift)	A Six Sigma index, PsiSigmaProbDefectShift calculates the Probability of Defect, with a shift, outside of the upper and lower limits. This statistic is defined as: $\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) +$ $1 - \delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaProbDefectShiftLower	sigmaProbDefecShiftLower(lower_limit, shift)	A Six Sigma index, PsiSigmaProbDefectShiftLower calculates the Probability of Defect, with a shift, outside of the lower limit. This statistic is defined as: $\delta^{-1}(\frac{LowerSpecificationLimit -\hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaProbDefecShiftUpper	sigmaProbDefecShiftUpper(upper_limit, shift)	A Six Sigma index, PsiSigmaProbDefectShiftUpper calculates the Probability of Defect, with a shift, outside of the upper limit. This statistic is defined as:

		$1-\delta^{-1}(\frac{UpperSpecificationLimit\ -\hat{\mu}}{\hat{\sigma}}-Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaUpperBound	sigmaUpperBound(number_stdev)	A Six Sigma index, PsiSigmaUpperBound calculates the Upper Bound as a specific number of standard deviations above the mean and is defined as: $\hat{\mu}-\hat{\sigma}*\#StandardDeviations$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaYield	sigmaYield(lower_limit, upper_limit, shift)	A Six Sigma index, PsiSigmaYield calculates the Six Sigma Yield with a shift, or the fraction of the process that is free of defects. This statistic is defined as: $\delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}}-Shift)-$ $\delta^{-1}(\frac{LowerSpecificationLimit-\hat{\mu}}{\hat{\sigma}}-Shift)$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaZLower	sigmaZLower(lower_limit): []	A Six Sigma index, PsiSigmaZLower calculates the number of standard deviations of the process that the lower limit is below the mean of the process. This statistic is defined as: $\frac{\hat{\mu}-LowerSpecificationLimit}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaZMin	sigmaZMin(lower_limit, upper_limit, [simulation])	A Six Sigma index, PsiSigmaZLower calculates the minimum of PsiSigmaZLower and PsiSigmaZUpper. This statistic is defined as: $\frac{MIN(\hat{\mu}-LowerSpecificationLimit,UpperSpecificationLimit-\hat{\mu})}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaZUpper	sigmaZLower(upper_limit): []	A Six Sigma index, PsiSigmaZUpper calculates the number of standard deviations of the process that the upper limit is above the mean of the process. This statistic is defined as: $\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
skewness skew	Skewness: [] Skew: []	Skewness returns the skewness for the specified uncertain function. Skewness is the 3rd moment of an uncertain function, and describes the asymmetry of its distribution. Skewness can be either positive or negative: Positive skewness implies that the distribution is right skewed (longer right tails), and negative skewness implies that the

		distribution is left skewed (longer left tails). Skewness is computed as: $skewness\left(X\right)=\frac{\sqrt{n}\sum_{i=1}^{n}\left(x_i-\mu\right)^3}{\left[\sum_{i=1}^{n}\left(x_i-\mu\right)^2\right]^{3/2}}$ where μ is the mean of the trial values.

spearmanRho	spearmanRho(unc_func_or_var): []
	Example where SpearmanRho statistic is used to return a correlation coefficient between two uncertain functions, uncFunc1 and uncFunc2.
	`uncertainFunctions: {`
	`  "uncFunc1": {"formula": "e17"},`
	`  "uncFunc2": {"formula": "d17",`
	`     "spearmanRho(uncFunc1)": []}`
		The SpearmanRho statistic returns a non-parametric measure (based on trial ranks). This function measures the correlation between two uncertain variables or functions. This statistic can be used to determine how (if at all) the two uncertain variables or functions are correlated.
		The Spearman correlation between two variables is equal to the Pearson correlation between the rank values of those two variables; while Pearson's correlation assesses linear relationships, Spearman's correlation assesses monotonic relationships (whether linear or not). If there are no repeated data values, a perfect Spearman correlation of +1 or −1 occurs when each of the variables is a perfect monotone function of the other.
		The Spearman correlation between two variables will be high when observations have a similar rank between the two variables or functions, and low when observations have a dissimilar rank between the two variables or functions.

stdDevCI	stdDevCI(confidence_level): []	StdDevCI returns the confidence ‘half-interval’ for the estimated standard deviation of the simulation trials (returned by the stdDev():[] function) for the specified uncertain function *cell*, at *confidence level* (for example 0.95 or 0.99). If σ is the value returned by stdDev():[] and δ is the value returned by stdDevCI():[], the true mean is estimated to lie within the interval σ - δ to σ + δ. If σ^2 (*n*) is the sample variance from *n* trial values, α = 1 – *confidence level*, and $t_{n-1,\ 1-\alpha/2}$ is the upper 1-α/2 critical point of the Student’s t-distribution with *n*-1 degrees of freedom, the confidence half-interval δ is computed as: $t_{n-1,1-\frac{\alpha}{2}}\sigma(n)\sqrt{\frac{k-1}{4(n-1)}}$ See also the description of the MeanCI() function.
stdev	stdev: []	StdDev returns the standard deviation for the specified uncertain function. Standard deviation is a measure of the dispersion of an uncertain function, and accounts for both positive and negative deviations from the mean. The square of standard deviation is the Variance. Standard deviation is defined as: $stddev\left(X\right)=\sqrt{E\left[X^2\right]-\left(E\left[X\right]\right)^2}$ The sampled population standard deviation is given by $stddev\left(X\right)=\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i-\mu\right)^2}$

		where E[.] is the expected value, and μ is the mean of the trial values. As a rough rule, about ¾ of the values of any uncertain variable are within two standard deviations from the mean. A large standard deviation indicates that most of the trial values are away from the mean, and a small standard deviation indicates that most of the trial values are close to the mean.
sterr	sterr: []	StdErr finds the standard error of the mean of the specified uncertain function. This function can be defined as the standard deviation of the sample mean ans is calculated as: $SE\overline{x} = \frac{s}{\sqrt{n}}$ where *s* is the sample standard deviation and *n* is the size of the sample.
target	target(target_value_[simulation]: []	Target returns the cumulative frequency of the target value in the distribution of trial values for the specified uncertain function. This function returns the proportion of simulated values for the uncertain function that are less than or equal to *target value*.
targetCI	targetCI(target_value, confidence_level)	TargetCI returns the confidence “half-interval” for the cumulative probability of the target value in a distribution of trial values for the specified uncertain function. This means that Target is accurate within Target +/- TargetCI with a given confidence level. This function is computed as: Lower: Target-TargetCI, Upper: Target + TargetCI. This function together with MeanCI, MeanCIB, StdDevCI, CITrials and the newly added PercentileCI, make up the confidence interval functions in Rason Services. Example: TargetCI (7, 0.99,2) - Finds the confidence half-interval for the uncertain function for the target value = 7, using a confidence level of 99% for the 2nd simulation.
targetD	targetD(target_value)	TargetD returns the descending cumulative probability of the target value in the distribution of trial values for the specified uncertain function. This function returns the proportion of simulated values for the uncertain function that are less than or equal to *target value*.
trials	trials: []	Returns the trial values of the uncertain function.
variance var	Variance: [] var: []	Variance returns the variance for the specified uncertain function. Like standard deviation, variance is a measure of the spread or dispersion of the distribution of trial values for the uncertain function, and takes into account both positive and negative deviations from the mean. The square root of variance is the standard deviation. The variance is the 2nd moment of the distribution of trials and is computed as: $\operatorname{var}(X) = E\left[X^2\right] - \left(E\left[X\right]\right)^2$ The sampled population variance is given by $\operatorname{var}(X) = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \mu\right)^2$

Uncertain Variables

The "uncertain variables" section is required in a simulation, simulation optimization or stochastic optimization model. Here you will define an uncertain variable or block of uncertain variables using a PSI distribution such as PSILogNormal or PSITriangular. In return you may ask for the final value and/or any of the twelve statistical values computed for the uncertain variables such as mean, standard deviation, variance, skewness, etc.

In the example code below, the uncertain variable `no_shows` follows the PsiLogNormal distribution with parameters mean = 5 and standard deviation = 1. In return, the expected mean of the function (`mean: []`) and the standard deviation (`stdev: []`) will be returned in the Result.

```
uncertainVariables: {
        no_shows: {
            formula: "PsiNormal(5, 1)",
            mean: [],
            stdev: []
        }
    },
```

We also could have created the uncertain variable `no_show` array by using an alternate syntax, shown below.

```
 uncertaionVariables : {
    { name: "no_shows",
      formula: "PsiNormal(5,1)" ,
      mean: [],
      stdev: []
    }

}
```

Note: Psi Distribution functions, such as PsiNormal(), PsiBeta(), etc., must be assigned to a single variable within the uncertainVariable section, in a RASON model. The use of Psi Distribution functions in an array is not supported.

Please see the table below for all input properties available in `uncertainVariables`.

Input Property	Example	Definition
`aliasName`	`aliasName: “variables”`	This property is automatically inserted into the converted RASON model when an Excel model is deployed through Analytic Solver’s Deploy Model button, if a block of cells containing uncertain variables is assigned a defined name in the Excel Solver model.
`comment`	`comment: "number of no-sho for a flight"`	Enter a comment here to describe the uncertain variable. (Optional)
`name`	`name: "no_shows"`	Enter a name for the uncertain variable. (Optional)
`dimensions`	5. `dimensions: [3,1]` 6. `dimensions: [3]` 7. `dimensions: [1,3]` 8. `dimensions: [3,2]`	5. Defines a 1 – dimensional vertical array with 3 rows and 1 column. 6. Defines a 1-dimensional vertical array with 3 rows and 1 column. 7. Defines a 2 – dimensional horizontal array with 1 row and 3 columns.

		8. Defines a 2 – dimensional array with 3 rows and 2 columns. All arrays are 1 – based. If creating a block of variables, you must use the dimensions property to define the size and shape of the array.
`formula`	`formula: "PsiNormal(5,1)"`	Use this property to define the Psi Distribution function used in the uncertain variable. To add cutoffs values, censor values, a correlation matrix, shift, etc, use the appropriate Psi function within this property, i.e. `formula: =PsiNormal(5,1,PsiTruncate(3, 7)).` See below for a list of all Psi Distributions and Psi Distribution function properties supported.

Psi Distribution Functions

The PSI Distribution functions are used to define the ‘nature of the uncertainty’ assumed by uncertain variables. They can be broadly classified into four groups:

- *Continuous analytic* distributions such as PsiUniform() and PsiNormal()
- *Discrete analytic* distributions such as PsiBinomial() and PsiGeometric()
- *Custom* distributions such as PsiCumul() and PsiGeneral()
- *Special* distributions such as PsiSip() and PsiSlurp()

On each trial of a simulation, Risk Solver Engine (RSE) draws a *random sample* value from each PSI Distribution function you use. PsiSip() and PsiSlurp() operate differently: On each trial, RSE draws the *next sequential* value listed in the SIP or SLURP for that uncertain variable. Then Risk Solver uses these sample values to calculate your model and its uncertain functions

The sample values drawn for PSI Distribution functions other than PsiSip() and PsiSlurp() depend on the *type* of distribution function, the *parameters* of the distribution (for example, mean and variance for the PsiNormal distribution), and the *property* functions that you pass as additional arguments to the distribution function call, which can shift, truncate, or lock the distribution, or correlate its sample values with samples drawn for other uncertain variables. To learn more about the analytic probability distributions supported by the RASON modeling language, see the Appendix.

Statistic Functions

An output property must be specified within the uncertain variable definition as an empty array. An output property for an uncertain variable is a statistic function such as mean or standard deviation. During a simulation of 1,000 trials, 1,000 random sample values will be drawn for the uncertain variable, and used to compute the statistic . Hence, you can think of the uncertain variable as ‘containing’ an array of 1,000 values. But the output property (i.e. mean) will contain *one* value, which is the average or mean of the 1,000 values computed for the uncertain variable. See the ta

Accessing Statistics from Different Simulations

Each PSI Statistic function is assigned to a *simulation* index. If you want statistic values from a specific simulation, say simulation #2, then you must set `simulationIndex: 2` in `modelSettings` as well as use the optional simulation index for the function. For more information, please see this option description in the *Model Settings* topic above.

A Note to Analytic Solver Users

- Selecting a specific simulation value within a statistic function is not supported. If interested on a particular simulation then users should use simulationIndex in modelSettings. Then output pertaining (only) to that simulation will be available in the results.
- PsiTheo functions are supported in RASON. However, the translation of PisTheo functions from Excel into RASON is currently not supported.
- Statistics in an Analytic Solver model may be translated into a Rason model using Analytic Solver's Create App functionality when function arguments are explicitly provided as constants or cell references, if the cell references point to constant values. For example, =PsiPercentile(cell, **A1**) where A1 = 0.5 or simply =PsiPercentile(cell, 0.5).

Output Property	Example	Definition
absDev	absDev:[]	AbsDev returns the average of the absolute deviations of the specified uncertain function's sample values from their mean. This is also known as Mean Absolute Deviation (MAD), especially in time series analysis applications. It is defined as: $MAD(X) = \frac{1}{n}\sum_{i=1}^{n}\lvert\mu - x_i\rvert$ Here μ is the mean of the sample values.
bvar	`bvar(confidence_level): []`	BVaR returns the Value at Risk for the specified uncertain function *cell* at the specified 'confidence level,' which is better described as a *percentile* – for example, 0.95 or 0.99. (The "B" stands for "Basel" or "building block," and is used to distinguish this function name from a function named PsiVar().) In finance applications, the Value at Risk is the maximum loss that can occur at a given confidence level. In a distribution of returns or profits, losses would lie at the "left end" of the distribution and would be represented by negative numbers and smaller percentiles (say 0.01 or 0.05). But it is customary in Value at Risk analysis to treat losses as positive numbers at the "right end" of the distribution. Consider a Normal distribution, bvar (*percentile*) would be converted as –percentile(1- *percentile*).
citrials	citrials(confidence_level, tolerance):[]	CITrials returns the estimated number of simulation trials needed to ensure that the specified uncertain function's sample mean value (returned by the PsiMean() function) lies within the confidence interval specified by *confidence level* (for example 0.95 or 0.99) and the half-interval size given by *tolerance*. Note that this number of trials is sufficient only to ensure that the single output value specified by *cell* lies within the confidence interval. To ensure that *N* output cells lie within confidence intervals at level 1 - α (e.g. 0.95 = 1 – 0.05), use a confidence level of α / *N*.
coeffVar	coeffVar: []	CoeffVar finds the coefficient of variation for the specified uncertain function. This function is defined as the ratio of the standard deviation to the mean and is calculated as: $Cv = \frac{\sigma}{\mu}$ This statistic measures the magnitude of the variability in relation to the mean of the population.

correlation	correlation: []
	example:
	`uncertainVariables: {`
	`  "uncVar1": {"formula":`
	`      "PsiNormal(10,5)"},`
	`  "uncVar2": {"formula":`
	`      "PsiNormal(20,10)",`
	`      "correlation(uncVar1)": []}`
		The correlation statistic returns the Pearson product moment correlation coefficient betweentwo uncertain variables or functions. Correlation is a measure of linear dependence between two uncertain variables or functions. The correlation coefficient can take on values between -1 and +1. A correlation of -1 indicates a perfect negative correlation (the cells move linearly in opposite directions); a correlation of +1 indicates a perfect positive correlation (the cells move linearly in the same direction). If the two random variables are independent, then their correlation coefficient is zero; but if the correlation coefficient is zero, this does not necessarily mean that the two variables are independent.
		Pearson's product moment correlation coefficient between random variables *X* and *Y* is defined as:

		$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-E[X])(Y-E[Y])]}{\sigma_X \sigma_Y}$$ In Monte Carlo simulation, this value is computed from the sample values *x*[] and *y*[] over *n* trials as: $$r_{x,y} = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{\sqrt{[n\sum_{i=1}^{n} {x_i}^2 - (\sum_{i=1}^{n} x_i)^2][n\sum_{i=1}^{n} {y_i}^2 - (\sum_{i=1}^{n} y_i)^2]}}$$
count	count(type): []	The count statistic returns the number of trials of the specified *type* executed in the most recent simulation for the uncertain function. The *type* argument may be 0 for all trials, 1 for normal or 'success' trials (where the calculated uncertain function resulted in a number), or 2 for error or 'failed' trials (where the calculated uncertain function resulted in an error value).
cvar	cvar(percentile):[]	CVaR returns the conditional Value at Risk for the specified uncertain function *cell*, at the specified 'confidence level' which is better described as a *percentile* – for example, 0.95 or 0.99. The conditional Value at Risk is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs. Like PsiBVaR, PsiCVaR returns a loss as a positive number. It is computed as the negative of the mean value of the specified uncertain function for the trials that lie between Min:[] and Percentile(1-*percentile*), inclusive.
expGain	expGain:[]	PsiExpGain returns the average of all positive data multiplied by 1 - percentrank of 0 among all data. It is always a positive number.
expGainRatio	expGainRatio: []	PsiExpGain returns the expected gain ratio for a specified uncertain function. This function is calculated as: $$ExpGaiRatio = \frac{ExpGain}{ExpGain + \lvert ExpLoss \rvert}$$ This value ranges between 0 and 1 inclusive.
expLoss	expLoss:[]	ExpLoss returns the average of all negative data multiplied by the percentrank of 0 among all data. It is always a negative number.
expLossRatio	expLossRatio:[]	ExpLoss returns the expected loss ratio for a specified uncertain function. This function is calculated as: This value ranges between 0 and 1 inclusive.
expValMargin	expValMargin:[]	ExpValMargin is calculated as: *ExpValMargin = ExpGainRatio – ExpLossRatio.* This statistic ranges between -1 and 1 inclusive.
frequency	frequency(frequency_type, bin bounds): [] example: frequency(0, -125000, -100000 100000,200000): []	Frequency returns an array of frequencies describing the distribution of trial values for the specified uncertain variable. Use this statistic to obtain the data required to draw a histogram chart in your application. The *freq_type* argument affects the contents of each element of the array result and can be 0, 1 or 2: 0 – Each element contains the frequency of trial values falling into the corresponding bin (like a probability density function)

		1 – Each element contains the cumulative frequency of trial values falling into the corresponding bin plus all lower bins (like a cumulative distribution function) 2– Each element contains the cumulative frequency of trial values falling into the corresponding bin plus all higher bins (like a reverse cumulative distribution function) In *RASON*, the *bin_bounds* argument is an array of values (e.g. { 5, 10, 15, 20 }) given in strictly increasing order. The number of elements in the array result will be one more than the number of values or cells in the *bin bounds* argument; the last element contains the number of trial values larger than the highest bin bound value.
indexValue	indexValue: []	Creates an empty array to hold the indexValue of the uncertain function.

kendallTau	kendallTau(unc_func_or_var): []
	Example where KendallTau statistic is used to return a correlation coefficient between two uncertain variables, uncVar1 and uncVar2.
	`uncertainVariables: {`
	`  "uncVar1": {"formula":`
	`      "PsiNormal(10,5)"},`
	`  "uncVar2": {"formula":`
	`      "PsiNormal(20,10)",`
	`      "kendallTau(uncVar1)": []}`
		The KendallTau statistic returns a non-parametric correlation coefficient (based on the relative ordering of ranks) between two uncertain variables or functions. This statistic can be used to determine how (if at all) the two uncertain variables or functions are correlated.
		The Kendall Tau rank correlation coefficient measures the ordinal association between two uncertain variables or functions. It is a measure of rank correlation. This correlation coefficient is high when observations have a similar rank between the two variables, and low when observations have a dissimilar rank between the two variables.

kurtosis kurt	kurtosis: [] kurt: []	Kurtosis returns the kurtosis for the specified uncertain function *cell*. Kurtosis is the 4th moment and measures the peakedness of the distribution of trial values. It is computed as: $kurtosis\left(X\right)=\frac{n\sum_{i=1}^{n}\left(x_i-\mu\right)^4}{\left[\sum_{i=1}^{n}\left(x_i-\mu\right)^2\right]^2}$ where μ is the mean of the trial values. A higher kurtosis indicates a distribution with a sharper peak and heavier tails, and that more of the variability is due to a small number of extreme outliers or values; a lower kurtosis indicates a distribution with a rounded peak and that more of the variability is due to many modest-sized values.
maximum max	maximum: [] max: []	Max returns the maximum value attained by the specified uncertain function over all the trials in the simulation.
mean average expectation	mean: [] average: [] expectation: []	Mean returns the mean value for the specified uncertain function. The mean or average value is the 1st moment of the distribution of trials and is computed as: $\mu=\frac{\sum_{i=1}^{n}x_i}{n}$

		The mean is frequently used as a measure of central tendency or the “middle” of an uncertain variable; but for skewed distributions, care must be taken in using the mean as a measure of central tendency because the mean is easily distorted by extreme outlier values.
meanCI	meanCI(confidence_level)	MeanCI returns the confidence “half-interval” for the estimated mean value (returned by mean: []) for the specified uncertain function, at the specified *confidence level* (for example 0.95 or 0.99). If μ is the value returned by mean:[] and δ is the value returned by meanCI(confidence_level):[], the true mean is estimated to lie within the interval μ - δ to μ + δ. The *confidence level* can be interpreted as follows: If we compute a large number of independent estimates of confidence intervals on the true mean of the uncertain function, each based on *n* observations with *n* sufficiently large, then the proportion of these confidence intervals that contain the true mean of the function should equal the *confidence level*. If $\sigma^2 (n)$ is the sample variance from *n* trial values, α = 1 – *confidence level*, and $t_{n-1,\ 1-\alpha/2}$ is the upper 1-α/2 critical point of the Student’s t-distribution with *n*-1 degrees of freedom, the confidence half-interval δ is computed as: $t_{n-1,1-\alpha/2}\sqrt{\frac{\sigma^2(n)}{n}}$ The confidence interval measures the precision with which we have estimated the true mean. Larger half widths imply that there is a lot of variability in our estimates. The above formula for the half-width assumes that the individual x_is are normally distributed; when this is not the case, the above formula still gives us an *approximate* confidence interval on the true mean of the uncertain function.
meanCIB	meanCIB(confidence_level, [lowerbound]):[]	MeanCIB returns the lower or upper bound of the confidence interval (half width) of the mean value for the specified uncertain function. *Confidence_level* – Enter the desired confidence level, i.e. 0.95 or 0.99. *lowerbound* – (Optional) Enter true for the lower (default) or false for the upper bound. *Example*: meanCIB(.99, true, 5) returns the lower bound for the 99% confidence interval for the distribution for simulation index 5.
median	median: []	Mean:[] returns the median value for the specified uncertain function. The median value is the 50th percentile of the distribution of trials and is computed as: $X_{(n+1)/2}$ if n is odd $\frac{X_{n/2} + X_{(n/2)+1}}{2}$ if n is even

		The median is a very useful statistic for measuring the center of a distribution.
minimum min	minimum: [] min: []	Min returns the minimum value attained by the specified uncertain function over all the trials in the simulation.
mode	mode: []	Mode returns the mode of the specified uncertain function. For discrete distributions, this is the most frequently occurring value (where the probability mass function has its greatest value). For continuous distributions, Mode() returns the *half-sample mode* as defined by D.R. Bickel, a robust estimator that is less sensitive to outliers than most other estimators of location.
percentileCI	percentile(percentile, confidence_level): []	PercentileCI returns the confidence “half-interval” for a given percentile (.01-.99) value for the specified uncertain function. This function is computed as: Lower: Percentile-PercentileCI, Upper: Percentile + PercentileCI. Since the output of PercentileCI is symmetric, the mean and median are *theoretically* the same, i.e. MeanCI(0.95) is expected to be approximately equal to PercentileCI(0.5, 0.95). This function together with MeanCI, MeanCIB, StdDevCI, CITrials and the newly added TargetCI, make up the confidence interval functions in RASON. Example: PercentileCI (0.95, 0.99,2):[] - Finds the confidence half-interval for the uncertain function using the 95th percentile and a confidence level of 99% for the 2nd simulation.
percentileD	percentileD(percentile): []	PercentileD returns a descending *percentile* (.01-.99) value for the specified uncertain function: This means that *m* (or *m*%) of the simulation trials have values less than the returned value, where *m* is the percentile.
percentiles	percentiles: [] percentile(confidence_level):[]	percentiles: [] returns all *percentile* (.01-.99) values for the specified uncertain function: This means that *m* (or 100*m*%) of the simulation trials have values less than the returned value, where *m* is the percentile. percentile(0.X) - Returns the specific percentile value. Values must be between 0.01 and 0.99.
range	range: []	Range returns the range of the specified uncertain function *cell*. The range is the difference between the maximum and minimum values attained in the distribution of trial values.
semiDev	semiDev(q, [target]): []	SemiDev returns the semideviation for the specified uncertain function, relative to the *target* if specified. If the *target* is omitted, the mean value is used. This is a *one-sided* measure of dispersion of values of the uncertain function. The semideviation is the square root of the semivariance, described directly below. If a *q* argument different from 2 is specified, SemiDev(): [] returns the *q*th root of the lower partial moment at power *q* of the uncertain function.
semiDev2	semiDev2([lowerdata]): []	SemiDev2 returns the standard deviation of the values in the distribution below or above the mean or the square root of SemiVar2.

		lowerdata – (Optional) Enter true for the lower (default) or false for the upper data. *Example*: semiDev2(true, 5) returns the standard deviation of the values below the mean for the distribution for simulation index 5.
semiVar	semiVar(q, target): []	SemiVar returns the semivariance for the specified uncertain function, if the argument *q* is omitted, or the 'lower partial moment' for the function, if an argument *q* different from 2 is specified. The semivariance is computed relative to the *target* if specified, or relative to the mean value if *target* is omitted. This is a measure of the dispersion of values of an uncertain function, but unlike the variance which measures (or penalizes) both positive and negative deviations from the target, the semivariance or lower partial moment is only concerned with *one-sided* deviations from the target. It is usually used in finance and insurance applications, when we are only concerned with downside risks (or loss in portfolio value). The semivariance is computed by summing only the downside differences from the target of all the trials, raised to the given power *q*, divided by the number of trials: $\frac{1}{n}\sum_{i=1}^{n}\left(t-x_i\right)_+^q$ $\left(x\right)_+ = \max\left(x,0\right)$ $t = \text{target value}$ All trials – not just the trials with downside deviations – are included in *n*. Again if *q* is different from 2, the result is called the 'lower partial moment.
semiVar2	sermiVar2([lowerdata]): []	SemiVar2 returns the variance of the values in the distribution below or above the mean. *lowerdata* – (Optional) Enter true for the lower (default) or false for the upper data. *Example*: =SemiVar2(true, 5) returns the variance of the values below the mean for the distribution for simulation index 5.
sigmaCP	sigmaCP(lower_limit, upper_limit): []	A Six Sigma index, PsiSigmaCP predicts what the process is capable of producing if the process mean is centered between the lower and upper limits. This index assumes the process output is normally distributed. $Cp = \frac{UpperSpecificationLimit - LowerSpecificationLimit}{6\hat{\sigma}}$ where $\hat{\sigma}$ is the estimated standard deviation of the process.
sigmaCPK	sigmaCPK(lower_limit, upper_limit,): []	A Six Sigma index, PsiSigmaCPK predicts what the process is capable of producing if the process mean is not centered between the lower and upper limits. This index assumes the process output is normally distributed and will be negative if the process mean falls outside of the lower and upper specification limits. $Cpk = \frac{MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

sigmaCPKLower	sigmaCPKLower(lower_limit, [simulation]): []	A Six Sigma index, PsiSigmaCPKLower calculates the one-sided Process Capability Index based on the lower specification limit. This index assumes the process output is normally distributed. $Cp, lower = \frac{\hat{\mu} - LowerSpecificationLimit}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaCPKUpper	sigmaCPKUpper(lower_limit): []	A Six Sigma index, PsiSigmaCPKUpper calculates the one-sided Process Capability Index based on the upper specification limit. This index assumes the process output is normally distributed. $Cp, upper = \frac{UpperSpecificationLimit - \hat{\mu}}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaCPM	sigmaCPM(lower_limit, upper_limit: []	A Six Sigma index, PsiSigmaCPM calculates the capability of the process around a target value. This index is referred to as the Taguchi Capability Index. This index assumes the process output is normally distributed and is always positive. $Cpm = \frac{\hat{C}p}{\sqrt{1+(\frac{\hat{\mu}-T}{\hat{\sigma}})^2}}$ where $\hat{C}p$ is the process capability (PsiSigmaCP), $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and T is the target process mean.
sigmaDefectPPM	sigmaDefectPPM(lower_limit, higher_limit)	A Six Sigma index, PsiSigmaDefectPPM calculates the Defective Parts per Million. $DPMO = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}}) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}})) * 1000000$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaDefectShiftPPM	sigmaDefectShiftPPM(lower_limit, upper_limit, shift): []	A Six Sigma index, PsiSigmaDefectShiftPPM calculates the Defective Parts per Million with an added shift. $DPMOShift = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)) * 1000000$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaDefectShiftPPMLower	sigmaDefectShiftPPMLower(lower_limit, shift)	A Six Sigma index, PsiSigmaDefectShiftPPMLower calculates the Defective Parts per Million, with a shift, below the lower specification limit. $DPMOshift, lower = (\delta^{-1}(\frac{LowerSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$

		where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaDefectShiftPPMUpper	sigmaDefectShiftPPMUpper(upper_limit)	A Six Sigma index, PsiSigmaDefectShiftPPMUpper calculates the Defective Parts per Million, with a shift, above the lower specification limit. $DPMOshift, upper = (\delta^{-1}(\frac{UpperSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$ where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaK	sigmaK(lower_limit, upper_limit): []	A Six Sigma index, PsiSigmaK calculates the Measure of Process Center and is defined as: $1 - \frac{2*MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{UpperSpecificationLimit - LowerSpecificationLimit}$ where$\hat{\mu}$ is the process mean.
sigmaLowerBound	sigmaLowerBound(number_stdev)	A Six Sigma index, PsiSigmaLowerBound calculates the Lower Bound as a specific number of standard deviations below the mean and is defined as: $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaProbDefectShift	sigmaProbDefectShift(lower_limit, upper_limit, shift)	A Six Sigma index, PsiSigmaProbDefectShift calculates the Probability of Defect, with a shift, outside of the upper and lower limits. This statistic is defined as: $\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) +$ $1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaProbDefectShiftLower	sigmaProbDefecShiftLower(lower_limit, shift)	A Six Sigma index, PsiSigmaProbDefectShiftLower calculates the Probability of Defect, with a shift, outside of the lower limit. This statistic is defined as: $\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaProbDefecShiftUpper	sigmaProbDefecShiftUpper(upper_limit, shift)	A Six Sigma index, PsiSigmaProbDefectShiftUpper calculates the Probability of Defect, with a shift, outside of the upper limit. This statistic is defined as:

		$1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaUpperBound	sigmaUpperBound(number_stdev)	A Six Sigma index, PsiSigmaUpperBound calculates the Upper Bound as a specific number of standard deviations above the mean and is defined as: $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaYield	sigmaYield(lower_limit, upper_limit, shift)	A Six Sigma index, PsiSigmaYield calculates the Six Sigma Yield with a shift, or the fraction of the process that is free of defects. This statistic is defined as: $\delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) - \delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.
sigmaZLower	sigmaZLower(lower_limit): []	A Six Sigma index, PsiSigmaZLower calculates the number of standard deviations of the process that the lower limit is below the mean of the process. This statistic is defined as: $\frac{\hat{\mu} - LowerSpecificationLimit}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaZMin	sigmaZMin(lower_limit, upper_limit, [simulation])	A Six Sigma index, PsiSigmaZLower calculates the minimum of PsiSigmaZLower and PsiSigmaZUpper. This statistic is defined as: $\frac{MIN(\hat{\mu} - LowerSpecificationLimit, UpperSpecificationLimit - \hat{\mu})}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
sigmaZUpper	sigmaZLower(upper_limit): []	A Six Sigma index, PsiSigmaZUpper calculates the number of standard deviations of the process that the upper limit is above the mean of the process. This statistic is defined as: $\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
skewness skew	Skewness: [] Skew: []	Skewness returns the skewness for the specified uncertain function. Skewness is the 3rd moment of an uncertain function, and describes the asymmetry of its distribution. Skewness can be either positive

		or negative: Positive skewness implies that the distribution is right skewed (longer right tails), and negative skewness implies that the distribution is left skewed (longer left tails). Skewness is computed as: $$skewness\left(X\right)=\frac{\sqrt{n}\sum_{i=1}^{n}\left(x_i-\mu\right)^3}{\left[\sum_{i=1}^{n}\left(x_i-\mu\right)^2\right]^{3/2}}$$ where μ is the mean of the trial values.

spearmanRho	spearmanRho(unc_func_or_var): []
	Example where SpearmanRho statistic is used to return a correlation coefficient between two uncertain variables, uncVar1 and uncVar2.
	`uncertainVariables: {`
	`  "uncVar1": {"formula":`
	`      "PsiNormal(10,5)"},`
	`  "uncVar2": {"formula":`
	`      "PsiNormal(20,10)",`
	`      "spearmanRho(uncVar1)": []}`
		The SpearmanRho statistic returns a non-parametric measure (based on trial ranks). This function measures the correlation between two uncertain variables or functions. This statistic can be used to determine how (if at all) the two uncertain variables or functions are correlated.
		The Spearman correlation between two variables is equal to the Pearson correlation between the rank values of those two variables; while Pearson's correlation assesses linear relationships, Spearman's correlation assesses monotonic relationships (whether linear or not). If there are no repeated data values, a perfect Spearman correlation of +1 or −1 occurs when each of the variables is a perfect monotone function of the other.
		The Spearman correlation between two variables will be high when observations have a similar rank between the two variables or functions, and low when observations have a dissimilar rank between the two variables or functions.

stdDevCI	stdDevCI(confidence_level): []	StdDevCI returns the confidence ‘half-interval’ for the estimated standard deviation of the simulation trials (returned by the stdDev():[] function) for the specified uncertain function *cell*, at *confidence level* (for example 0.95 or 0.99). If σ is the value returned by stdDev():[] and δ is the value returned by stdDevCI():[], the true mean is estimated to lie within the interval σ - δ to σ + δ. If σ^2 (*n*) is the sample variance from *n* trial values, α = 1 – *confidence level*, and $t_{n-1,\,1-\alpha/2}$ is the upper 1-α/2 critical point of the Student’s t-distribution with *n*-1 degrees of freedom, the confidence half-interval δ is computed as: $$t_{n-1,1-\frac{\alpha}{2}}\sigma(n)\sqrt{\frac{k-1}{4(n-1)}}$$ See also the description of the MeanCI() function.
stdev	stdev: []	StdDev returns the standard deviation for the specified uncertain function. Standard deviation is a measure of the dispersion of an uncertain function, and accounts for both positive and negative deviations from the mean. The square of standard deviation is the Variance. Standard deviation is defined as:

		$stddev(X)=\sqrt{E\left[X^2\right]-\left(E\left[X\right]\right)^2}$ The sampled population standard deviation is given by $stddev(X)=\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\mu)^2}$ where E[.] is the expected value, and μ is the mean of the trial values. As a rough rule, about ¾ of the values of any uncertain variable are within two standard deviations from the mean. A large standard deviation indicates that most of the trial values are away from the mean, and a small standard deviation indicates that most of the trial values are close to the mean.
sterr	sterr: []	StdErr finds the standard error of the mean of the specified uncertain function. This function can be defined as the standard deviation of the sample mean ans is calculated as: $SE\overline{x}=\frac{s}{\sqrt{n}}$ where *s* is the sample standard deviation and *n* is the size of the sample.
target	target(target_value_[simulation]: []	Target returns the cumulative frequency of the target value in the distribution of trial values for the specified uncertain function. This function returns the proportion of simulated values for the uncertain function that are less than or equal to *target value*.
targetCI	targetCI(target_value, confidence_level)	TargetCI returns the confidence “half-interval” for the cumulative probability of the target value in a distribution of trial values for the specified uncertain function. This means that Target is accurate within Target +/- TargetCI with a given confidence level. This function is computed as: Lower: Target-TargetCI, Upper: Target + TargetCI. This function together with MeanCI, MeanCIB, StdDevCI, CITrials and the newly added PercentileCI, make up the confidence interval functions in Rason Services. Example: TargetCI (7, 0.99,2) - Finds the confidence half-interval for the uncertain function for the target value = 7, using a confidence level of 99% for the 2nd simulation.
targetD	targetD(target_value)	TargetD returns the descending cumulative probability of the target value in the distribution of trial values for the specified uncertain function. This function returns the proportion of simulated values for the uncertain function that are less than or equal to *target value*.
theoKurtosis*	theoKurtosis:[]	Returns the analytic kurtosis (4th moment) value for the specified distribution.
theoMax*	theoMax:[]	Returns the maximum value of the specified distribution.
theoMean*	theoMean:[]	Returns the mean of the specified distribution.
theoMedian*	theoMedian:[]	Returns the median of the specified distribution.
theoMin*	theoMin:[]	Returns the minimum of the specified distribution.
theoMode*	theoMode:[]	Returns the mode of the specified distribution.

theoPercentile* theoPtoX	theoPercentile(percentile):[] theoPtoX(percentile):[]	The functions theoPercentile and theoPtoX are alternate names for the same function. Both functions return the analytic percentile (CDFInv) value for the specified distribution specified. Enter the desired percentile (in decimal form) for the `percentile` argument, i.e. .01, .30, or .98. The percentile value must be between 0 and 1.
theoPercentileD* theoQtoX	theoPercentileD(percentile):[] theoQtoX(percentile):[]	The functions theoPercentileD and PsiTheoQtoX are alternate names for the same function. Both return the percentile (CDFInv descending) value for the specified distribution. Enter the desired percentile (in decimal form) for the `percentile` argument, i.e. .01, .30, or .98. The percentile value must be between 0 and 1.
theoRange*	theoRange:[]	Returns the range information for the specified distribution.
theoSkewness*	theoSkewness:[]	Returns the skewness of the specified distribution.
theoStdDev*	theoStdDev:[]	Returns the standard deviation of the specified distribution.
theoTarget* theoXtoP	theoTarget(target):[] theoXtoP(target):[]	Returns the cumulative probability for `target` for the specified distribution. The cumulative probability returned is the probability of a value less than or equal to `target` occurring. (The functions theoTarget and theoXtoP are alternative names for the same function.)
theoTargetD* theoXtoQ	theoTargetD(target):[] theoXtoP(target):[]	Returns the cumulative descending probability for `target` for the specified distribution. The cumulative probability returned is the probability of a value greater than or equal to `target` occurring. (The functions theoTargetD and theoXtoQ are alternative names for the same function.)
theoVariance*	theoVariance:[]	Returns the variance of the specified distribution.
theoXtoY*	theoXtoY(value):[]	Returns the probability for value for the specified distribution. For a continuous distribution, the value returned is the probability density value at `value`. For a discrete distribution, the value returned is the probability value at `value`.
trials	trials: []	Returns the trial values of the uncertain function.
variance var	Variance: [] var: []	Variance returns the variance for the specified uncertain function. Like standard deviation, variance is a measure of the spread or dispersion of the distribution of trial values for the uncertain function, and takes into account both positive and negative deviations from the mean. The square root of variance is the standard deviation. The variance is the 2nd moment of the distribution of trials and is computed as: $\operatorname{var}(X) = E\left[X^2\right] - \left(E[X]\right)^2$ The sampled population variance is given by $\operatorname{var}(X) = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \mu\right)^2$

*Theo Functions are statistics functions that return a statistic on a simulation input distribution, or uncertain variable. If a theoXXX function is applied to an output function or if the statistic can not be computed, the function will be ignored and will not appear in the results. The theoXXX functions compute the moment only when distribution parameters are not dependent on other distributions or decision variables in order to guarantee that the moment is constant throughout solving. These functions were designed to aid in the visualization of results and for comparison of the exact analytic moment with the trial statistics.

A Note on Excel Ranges in a Converted RASON Model

When a model built using Analytic Solver, in Destop Excel or Excel Online, is converted to the RASON modeling language using Create App, you'll notice that many RASON components, such as names for variables/constraints, uncertain variables/functions/, the objective function, etc. These Excel ranges are defined as in Excel using the row and column notations[a1 or A1:B2]. These ranges behave as they do in Excel. The Excel range, A1:C3, is a double array containing 9 elements. To access the contents of B2, you would use B2. Extreme care should be taken when removing these ranges from a converted model as inadvertent errors may result. For example, assumg the data definition "A1:C3" is changed to simply "A1C3". If cell "B2" is defined in another RASON component, an error will be generated since B2 references a value within "A1:C3" but not "A1C3".

Rason Data Mining Model Components

Introduction

This section introduces each of the nine components or sections which make up a data mining RASON model: "data", "datasources", "datasets", "weakLearner", "estimator", "transformer", "actions", "model" and "preProcessor". This chapter explains how each component of your model should be defined.

All algorithms featured in Analytic Solver and XLMiner SDK can be expressed using a standardized structure in RASON DM. This basic structure includes four major "sections" or "segments": datasources, datasets, estimator/transformer, and actions.

```
{
   "datasources": {},
   "datasets":{},
   "estimator"/"transformer": {
      "type":"",
      "algorithm":"",
      "parameters":",
   },
   "actions":{}
}
```

- datasources -- The "datasources" section is where the data for the model is acquired. Most times, the data is contained in an external data source such as a database or delimited file.
- datasets -- The "datasets" section is where the external dataset is "bound" to a RASON dataset.
- estimator/transformer – These sections are mutually exclusive -- A model may not contain both a transformer and an estimator. An "estimator" object *estimates* a model from the training data and stores the fitted model, which may be used later. Examples of estimators are classification or regression algorithms. These algorithms fit a model which can be used later to score new data. A "transformer" applies to estimators that do not fit a model but rather transform data, such as Feature Selection or Sampling.
- actions -- The function of the estimator or transformer is carried out within the "actions" section. If a model was "fit" within the estimator section, then the model is applied to the desired dataset (training, validation, test partitions or to new data) within this section. If a transformer was initiated, then the actual data transformation will be performed within this segment.

Rason DM also features several additional optional sections that can be used to further refine the Rason model. These additional sections are: data, model, preprocessor and weakLearner.

- data -- Data arrays may be defined and calculated in this optional section, to be used later in a data mining method. Scalars, arrays or tables containing scalars maybe be defined in the data section. If pulling data from an external source, this section may be used to "bind" the data to an array or table.
- model -- Used (only) when scoring a model. This section is similar to "datasets" but rather than refining imported data, this section defines a model that you can bind to when performing an "action" such as "forecast", "predict", "fit" or "transform".

  Note: Currently it is not possible to manually match features in the dataset with features in the new data, as is possible in Excel via the Scoring dialog. *Rather, features are always matched sequentially.*

- **preProcessor -- This optional section may be used for preliminarily data preparation or to compute values of some properties, which are passed later, at parse-time, to the RASON DM engine. This section is parsed once, before the model is parsed.**
- weakLearner -- This section is only required when a bagging or boosting estimator is specified in "estimator", and is used to define the weak learner used in these algorithms.

It is important to note that order inside the Rason model is *very* important as the Rason interpreter does *not* parse the model to determine the correct order beforehand. Therefore, "actions" may not appear before "estimator", "estimator" may not appear before "datasets" and so on.

Data

Data arrays may be defined and calculated in this optional section to be used later in a data mining method. If you are pulling data from an external source, use this section to "bind" the data to an array or table.

In the example code below, data from the `qty` column from the `parts_data` data source is assigned to the `parts` table. Note: A table is created here, rather than an array, by the use of the `valueCol` property.

```
"data": {
        "parts": {
            "binding": "parts_data", "valueCol": "qty"
        },
}
```

Scalars, arrays or tables containing scalars maybe be defined in the data section.

The following is an example of a scalar constant, which is neither an array nor a table.

```
"time": { "value": 10 }
```

In the example below, the array `wine` with size equal to 3 contains the values, A, B, and C. In this instance, the `binding` property allows write access to the `profit` array outside of the model environment using the keyword "get".

```
"wine": {
            "dimensions": [3], "value": [A, B, C], "binding": "get" },
},
```

To change the array elements in "wine" to C, D, E; you can pass new data directly in the REST API call, via standard HTTP GET parameters, for example:

```
$.get(https://rason.net/api/optimize?wine=C,D,E...
```

To change only one element, say the middle element from B to D, your call to the REST API, via standard HTTP GET parameters would change to:

```
$.get(https://rason.net/api/optimize?wine[2]=...
```

We also could have created the profit array by using an alternate syntax, shown below. However, when a parameter is defined in this way, you will not be able to pass new values to the array outside of the RASON model environment (as shown above).

```
"data" : [
  { "name": "wine", "value": [A, B, C], "binding": "get" }
 ],
```

All properties available for `data`, can be found in the table below.

Data Property	**Type**	**Explanation**
`name`	`"name": "parts"`	Use this property to define the table, array or scalar name.

`dimensions`	`"dimensions": [3,1]`
	`"dimensions": [3]`
	`"dimensions": [1,3]`
	`"dimensions": [3,2]`
		Defines a 1-dimensional vertical array.
		Defines a 1-dimensional vertical array.
		Defines a 2 – dimensional horizontal array with three elements.
		Defines a 2 – dimensional array or table with 3 rows and 2 columns.
		All arrays are 1 – based. If missing, array shape will be defined by the shape of the value property; however, for easier readability of the code, the use of the *dimensions* property is recommended.

`value`	`"value": [1, 1, 1]`
	`"value": [[1, 1, 1],`
	`[2, 2, 2],`
	`[3, 3, 3]]`
		Sets the values of the array.
		Sets the values of a table.
		If dimensions property is missing, the shape of the variable array will be determined by the shape of the value property. However, it is recommended that the *dimensions* property be used for readability purposes.

`valuecol`	`"valueCol": ["initials"]`	Used with `binding` property to bind imported values from a readable data source. If omitted, the RASON interpreter assumes the last column in the table as the input to `valueCol`.

`binding`	`"binding": "get"`
	`"Profit": { "binding": "profit_data" }`
		Allows data to be edited outside of the model from a URL or when calling the RASON^{TM} interpreter to solve an optimization or simulation model.
		Used to bind imported table from the `profit_data` datasource to a new table named `profit`.

`comment`	`"comment": "partsReq array holds the number of parts required to produce each product"`	Enter a comment here to describe the data.

Datasources ("datasources")

External data sources are defined in this section. Data from these sources is imported into parametric tables or arrays to be used in a data mining model. Currently the RASON modeling language supports ten different data sources: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "odata" (OData database), "mssql" (Microsoft Sequel), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file). Data sources such as "Access", "ODBC", "CSV", etc, contain data in tables with records described by index and value columns. Binding to these data sources results in table objects. Data source types such as Excel and CSV may contain data in 2-dimensional arrays without any descriptions. Binding to these data sources results in array objects. Objects are bound to data sources within the `data` section.

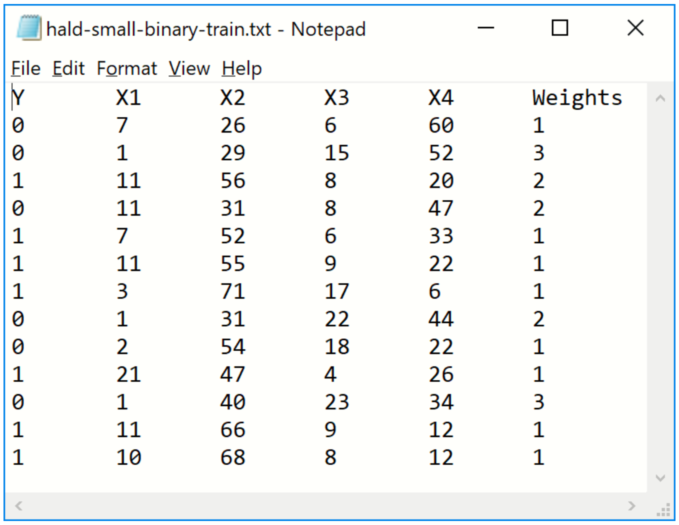
Importing

In the example below, three data sources are defined: "myTrainingData", "myValidationData" and "myTestData". In the first data source, myTrainingData, data is imported from the hald-small-binary-train.txt file. In the second data source, myValidationData, data is imported from hald-small-binary-valid.txt and in the third data source, myTestData, data is imported from hald-small-binary-test.txt.

The first property, `type`, specifies the type of file where the data is contained. In this example, the file is a CSV (Comma Separated Values) file. (A screenshot of hald-small-binary-train.txt is shown below the example code. The remaining files, hald-small-binary-validation.txt and hald-small-binary-test.txt are similar. The second property, `connection`, specifies the file name and location within quotes (`"/datafiles/hald-small-binary-train.txt"`). The third property `"direction": "import"` tells the RASON Server that the contents of each data source will be imported. (This is the default setting for this property.)

```
"datasources": {
      "myTrainingData": {
            "type": "csv",
            "connection": "/datafiles/hald-small-binary-train.txt",
            "direction": "import"
      }
      "myValidationData": {
            "type": "csv",
            "connection": "/datafiles/hald-small-binary-valid.txt",
            "direction": "import"
      }
      "myTestData": {
            "type": "csv",
            "connection": "/datafiles/hald-small-binary-test.txt",
            "direction": "import"
        }

},
```

hald-small-binary-train.txt CSV file


hald-small-binary-train.txt - Notepad
File Edit Format View Help

```
Y         X1        X2        X3        X4        Weights
0         7         26        6         60        1
0         1         29        15        52        3
1         11        56        8         20        2
0         11        31        8         47        2
1         7         52        6         33        1
1         11        55        9         22        1
1         3         71        17        6         1
0         1         31        22        44        2
0         2         54        18        22        1
1         21        47        4         26        1
0         1         40        23        34        3
1         11        66        9         12        1
1         10        68        8         12        1
```

In this next example, data is imported from an Excel table.

```
"datasources": {
      "msExcelSrc": {
            "type": "excel",
            "connection": "hald.xlsx",
            "selection": "Data!A2:E14",
            "headerExists": "true",
            "direction": "import"
      }
  },
```

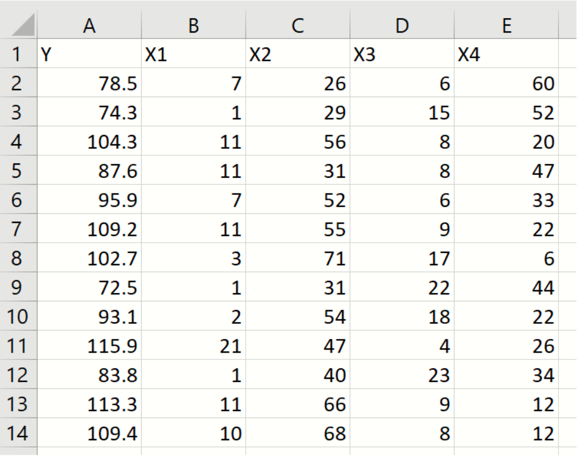
The first property, `type`, specifies that the data is contained in an Excel file. The second property, `connection`, specifies the name of the file, `"hald.xlsx"`.

The 3rd property, `selection: "Data!A2:E14"`, specifies the Excel range where the data is contained. Alternatively, we could also pass `"selection": "Data_Table"` where `"Data_Table"` is an Excel defined name given to the Excel Range, Data!A1:E14. The 4th property, `headerExists (`may also use simply `"header")` indicates whether the columns contain titles, or headers, or not, and is set to True by default.

Note: Column headings are contained in cells Data!A1:E1 and the data is contained in cells Data!A2:E14. Since headerExists is set to true, the default, RASON assumes the headings are contained in the next row up, which in this case in Row 1. If headerExists is set to false, the selection property would still be "selection":"Data!A2:E14"; the row containing the headings is never passed.

The 5th property, `"direction": "import",` indicates that the contents of hald.xlsx will be imported.

hald.xlsx Excel file



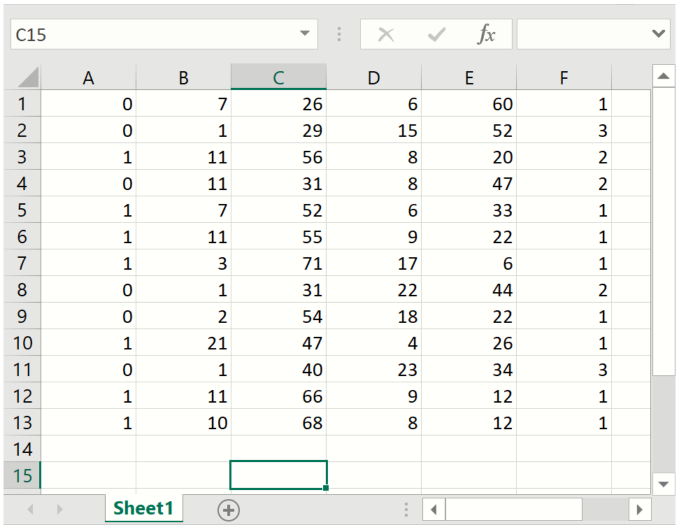
	A	B	C	D	E
1	Y	X1	X2	X3	X4
2	78.5	7	26	6	60
3	74.3	1	29	15	52
4	104.3	11	56	8	20
5	87.6	11	31	8	47
6	95.9	7	52	6	33
7	109.2	11	55	9	22
8	102.7	3	71	17	6
9	72.5	1	31	22	44
10	93.1	2	54	18	22
11	115.9	21	47	4	26
12	83.8	1	40	23	34
13	113.3	11	66	9	12
14	109.4	10	68	8	12

In this next example, data is imported using the "colIndex" and "rowIndex" properties. Recall that colIndex and rowIndex create a dataframe, rather than a RASON Table.

```
"datasources" : {
      "msExcelSrc":  {
            "type": "excel",
            "connection": "HaldRaw.xlsx",
            "selection": "Sheet1!A1:F13",
            "colIndex": "features",
            "rowIndex": "records",
            "direction": "import"
      }
}
```

As in the example above, the first property, `type`, specifies that the data is contained in an Excel worksheet, `"excel"`; the second property, `connection`, passes the name of the Excel file, `"HaldRaw.xlsx"`; and the third property, `selection`, passes the Excel cell range that contains the data, in this instance, `"Sheet1!A1:F13"`.

HaldRaw.xlsx Excel file



	A	B	C	D	E	F
1	0	7	26	6	60	1
2	0	1	29	15	52	3
3	1	11	56	8	20	2
4	0	11	31	8	47	2
5	1	7	52	6	33	1
6	1	11	55	9	22	1
7	1	3	71	17	6	1
8	0	1	31	22	44	2
9	0	2	54	18	22	1
10	1	21	47	4	26	1
11	0	1	40	23	34	3
12	1	11	66	9	12	1
13	1	10	68	8	12	1
14						
15						

However in this example, the 4th property, `colIndex`, binds the index name `features` to the columns and the 5th property, `rowIndex`, binds the index name `records` to the rows. The property `colIndex` binds a set of integers from 1 to the number of columns and the property `rowIndex` binds a set of integers from 1 to the number of rows to the 2-dimensional array . The 5th property, `"direction": "import",` indicates that the contents of haldraw.xlsx will be imported.

As a result, if a new product or new part is added, there will be no changes required to this section of the model. It is completely scalable.

This next example illustrates how to import data from an SQL database residing on an Azure server in the Cloud using an ODBC connection string.

```
"datasources": {
      "mssqlSrc": {
            "type": "mssql",
            "connection": "Server=Test-
            DELL;Database=Test;trusted_connection=Yes",
            "selection": "SELECT * FROM dbo.Data",
            "direction": "import"
      }
  },
```

The first property, `type`, specifies the type of file containing the data, in this case the file is an Microsoft SQL database. The second property, `connection`, passes the connection string as obtained from the server. The third property, `selection`, performs an SQL query from `dbo.Data` (<db_name>.<table_name>).

The next example illustrates how to import data from an OData data source. This model is also completely scalable. For more information on OData, see http://www.odata.org. Note: OData data sources are not currently writeable due to limitations in the common OData specification.

```
"datasources": {
      "odataSrc": {
            "type": 'odata',
            "connection": 'http://localhost:12345/',
            "selection": "Hald?$select=TriAlum,TriSil,TetraAlumFer,DicSil,
             Heat,Comment", "direction"="import"
      }
```

```
  }
```

The first property for `odataSrc`, `type`, specifies the type of file containing the data. In this case, the type is an OData data source. The second property, `connection`, specifies the location of the OData data source on the internet or distributed server. The third property, `"selection"`, imports six fields from the Hald table`,` `TriAlum, TriSil, TetraAlumFer, DicSil, Heat` and `Comment`.

In this example, the last selection property, `$format=json`, is not passed. This property stipulates the type of OData format (JSON or XML) in which the table should be returned. This is an optional argument. If passed, the OData service will return the data in the format specified, $format=json for JSON or $format=atom for XML. If omitted, the OData service will return the data in preferred format: JSON or XML. The RASON server will automatically recognize the format if not specified.

Using a Named Data Connection

In previous versions of RASON, models that accessed external databases required actual credentials to be passed, such as database URLs, port numbers, usernames, and passwords, in the text of the RASON model, in a dataSource declaration, as shown above and in the example code below.

<u>Previous versions of RASON</u>

```
 "parts_data": {
      "type": "odbc",
      "connection": "Driver={SQL Server Native Client
   11.0};Server=tcp:solver.database.windows.net,1433;Database=Rason;Uid=ra
   sonread;Pwd=Rason1234;Encrypt=yes;TrustServerCertificate=no;Connection
   Timeout=30;",
      "selection": "SELECT Parts as parts, Products as prods, Qty as qty
   FROM Parts ORDER BY ID",
      "indexCols": [ "parts", "prods" ],
      "valueCols": [ "qty" ],
      "direction": "import"
       },
```

RASON 2020 offers an alternative to tackle this security risk by substituting

```
"connection": "Driver={SQL Server Native Client
   11.0};Server=tcp:solver.database.windows.net,1433;Database=Rason;Uid=ra
   sonread;Pwd=Rason1234;Encrypt=yes;TrustServerCertificate=no;Connection
   Timeout=30;",
```

with three options: a file containing the contents of "connection" as in (1) below, a named Data Connection as shown in (2) or a URL pointing to Microsoft Common Data Service as shown in (3).

4. `"connection":  "File = filename",`

   RASON 2020 will interpret this as (i) get the text contents of `filename`, which must be attached to the current model instance and (ii) substitute this text for the string `"File=filename"`. Therefore, if `filename` contains the text `"Driver={SQLServerNativeClient…Timeout=30;"`, the effect will be the same as in previous versions of RASON.

5. `"connection": "Name=myname",` where myname is the name given to the Data Connection. See below for instructions on how to create a named Data Connection.

6. `"connection": "secret=uri",` where uri is the Microsoft Common Data Service URL

   `"connection": "xxxx.crm.dynamics.com"` where the actual Microsoft Common Data Service URL is passed directly to "connection".

   If using a with `"secret=url"` in the dataSources section of your RASON model, enter a URI of the form https://subdomain.crm.dynamics.com. , i.e.

"https://www.cdatacloud.net/solver/api.rsc/GoogleSheets1_ODataSourceExample_Sheet1",

RASON 2020 will interpret this as (i) get the text contents of the "secret" represented by the URL and (ii) substitute this text for the string "Secret=url". So if the "secret" contains the text "Driver={SQLNativeClient…Timeout=30;", the effect will be the same as in previous versions of RASON.

Similarly, if using CData Cloud Hub with "connection": "xxxx.crm.dynamics.com", enter a URI of the form https://subdomain.crm.dynamics.com.

RASON 2020 will interpret this as (i) get the text contents of the connection represented by the URL and (ii) substitute this text for the string "connection".

Currently, RASON 2020 supports "secrets" maintained, only, in an Azure Key Vault. Enterprise customers can provision their own Key Vault and arrange to authenticate the RASON Server to this Key Vault if so desired.

For more information on how to setup and maintain a named Data Connection, see the RASON Services WEB IDE chapter within the RASON User Guide.

See the table below for more examples illustrating how to import data in Rason DM.

Importing From:	

JSON file	"datasources": {
	   "jsonSrc": {
	      "type": "json",
	      "connection": "hald-small-nested.json",
	      "selection": "test.data",
	      "direction": "import"
	   }
	},

ODBC Database	"datasources": {
	   "odbcSrc": {
	      "type": "odbc",
	      "connection": "Driver={SQL Server Native
	       Client 11.0};Server=Frontline;Database=Test;
	       trusted_connection=Yes;
	       Connection Timeout=30;",
	      "selection": "SELECT * FROM dbo.Data",
	      "direction": "import"
	    }
	},

MS Access Source	"datasources": {
	   "msAccessSrc": {
	      "type": "access",
	      "connection": "ms-access-db.accdb",
	      "Selection": "SELECT * FROM Hald",
	      "direction": "import"
	   }
	},

Exporting

Evaluation results may either be 1. A part of the RASON response or 2. Bound to a writeable datasource. In the example below, "fittedModelJson" and "regressionSummary" are part of the RASON response while "influenceDiagnostics" is bound to the writeable datasource "myExportSrc". To view this complete example, see LinearRegression.json on the Try It page on RASON.com. Note: Some code has been removed from the example below for simplicity.

```
mlr: {
  comment: 'regression: linear model',
  datasources: {
    myTrainSrc: {
      type: 'csv',
      connection: 'hald-small-train.txt',
      direction:"import"
    },
    …
    myExportSrc: {
      type: 'csv',
      connection: 'Results/Export/influence-diagnostics.csv',
      direction:"export"
    }
  },
  datasets: {
    myTrainData: {
      binding: 'myTrainSrc',
      targetCol: 'Y'
    },
    …
  },
  estimator: {
    mlrEstimator: {
      type: 'regression',
      algorithm: 'linearRegression',
      parameters: {
        fitIntercept: true
      }
    }
  },
  actions: {
    mlrModel: {
      trainData: 'myTrainData',
      estimator: 'mlrEstimator',
      binding: 'myJSONSrc',
      action: 'fit',
      evaluations: [
        'fittedModelJson',
        {
          name: 'influenceDiagnostics',
          binding: 'myExportSrc'
        },
        'regressionSummary'
        …
      ]
      …

  }

}
```

Notes on exporting to a writable data source.

1. **"Type": "CSV" and "Type": "JSON" simply create or overwrite the files with the dataframe/table evaluation.**
2. **The "selection" property specifies the Excel worksheet and is optional when "Type":"Excel". If not provided, the worksheet name will be automatically assigned based on the rason script's name, action, and evaluation, i.e mlr-mlrmodel-influenceDiagnostics.**

3. **The "selection" property specifies the Database table name and is optional for all database types. If not provided, the table name will be automatically assigned as in 2 above.**
4. **Users can write to the same Excel workbook or same database – adding new worksheets/tables with subsequent evaluations.**
5. **It's also possible to create a new MS Access database file and write evaluations there.**
6. **Creating a new database for MS SQL/Oracle types or when using an ODBC connection string is not supported. As a result, "connection" must point to an existing database.**
7. See the table below for more examples illustrating how to import data in Rason DM.
8. Importing data and exporting results to the same data source is not supported when using the data mining solve endpoint. This is only supported when solving an optimization or simulation model where the initial variable values are imported and the final variable values are exported (to the same data source).

See the table below for more examples on exporting datamining/forecasting results to a writeable file.

Importing From:	

CSV File	myExportSrc: {
	  type: 'csv',
	  connection: 'Results/Export/influence-diagnostics.csv',
	  direction: "import"
	}

JSON File	myExportSrc: {
	  type: 'json',
	  connection: 'Results/Export/influence-diagnostics.json',
	  direction: "import"
	}

Excel Workbook	myExportSrc: {
	  type: 'excel',
	  connection: 'Results/Export/excel-export.xlsx',
	  selection: 'InfluenceDiagnosticsWorksheet',
	  direction: "import"
	}

Access Database	myExportSrc: {
	  type: 'access',
	  connection: 'Results/Export/access-export.accdb',
	  selection: 'InfluenceDiagnosticsTable',
	  direction: "import"
	}

MSSQL Database	myExportSrc: {
	  type: 'mssql',
	  connection: 'Server=OLEG-DELL;Database=test;trusted_connection=Yes',
	  selection: 'InfluenceDiagnosticsTable',
	  direction: "import"
	}

ODBC Connection	myExportSrc: {
	  type: 'odbc',
	  connection: 'Driver={SQL Server Native Client 11.0};Server=OLEG-DELL;Database=test;trusted_connection=Yes;Timeout=30;',
	  selection: 'InfluenceDiagnosticsTable',
	  direction: "import"
	}

All properties available for `dataSources`, can be found in the table below.

Data Source Property	**Example**	**Explanation**
`Binding`	`"binding":"mySrc"`	Binds dataset to new data source.
`colIndex`	`"colIndex": "prods"` This property creates a dataframe and should only be used in conjunction with rowIndex, not indexCols and valueCols which create a RASON table.	Use this property to create an implicit index set consisting of integer numbers from 1 to the number of columns. This property should be used when importing data not organized as a table, and thus not having index columns or value columns.
`connection`	`"connection":  "ProductMix.xlsx"`	Use this property to pass the filename of the data source. Note: Columns are assumed to have headers. If no header exists, set "headerExists" : false. When referring to excel ranges with "headerExists": true/false, do not include the header row in the range.
`content`	`"content": "corpus"` **Parameter Options** • `corpus` • `itemset` • `json-model` • `pmml-model` • `table` • `time-series`	Use this property to read data in some specific manner such as: corpus – text corpus itemset – item list json-model – Model in JSON format pmml-model – Model in PMML format table – table time-series – time series dataset

`direction`	`direction: "import"`
	`direction: "export"`
		Use `direction: "import"` when importing the contents of a file. Use `direction: "export"` when exporting results.

`headerExists`
`header`
	`"headerExists": true`
	`"header": true`
		Set to True by default. Parameter indicates if the data file contains column headings (true) or not (false).
		When referring to excel ranges with "headerExists": true/false, do not include the header row in the range.

`indexCols`	`"indexCols": ["parts", "prods"]` This property creates a RASON table and should only be used in conjunction with valueCols, not colIndex and rowIndex which create a dataframe.	Used in conjunction with `valueCols.` Use this property to index by dimension(s).

`rowIndex`	`"rowIndex": "parts"` This property creates a dataframe and should only be used in conjunction with rowIndex, not indexCols and valueCols which create a RASON table.	Use this property to create an implicit index set consisting of integer numbers from 1 to the number of rows. This property should be used when importing data not organized as a table, and thus not having index columns or value columns.

`selection`	`Importing`
	`1a. "selection": "Sheet1!B2:D6"`
	`1b. "selection": "Sheet1!Parts_Table"`
	`2.  "selection": "SELECT Parts, Products, Qty FROM Parts ORDER BY ID"`
	`Exporting`
	`1. "selection": "InfluenceDiagnosticsSheet"`
	`2. "selection": "InflusenceDiagnosticsTable"`
		Use this property to select the columns/fields to import.
		3. If data source is an Excel file, pass A. the Excel Range or B. an Excel defined name.
		4. If data source is an odbc database use: `SELECT` + desired fields separated by commas + `FROM` + name of table containing desired field(s) + `ORDER BY` + field name containing order index.
		When exporting use this property to:
		1. Specify the Excel worksheet. This property is optional when "Type":"Excel". If not provided, the worksheet name will be automatically assigned based on the rason script's name, action, and evaluation, i.e mlr-mlrmodel-influenceDiagnostics.
		2. Specify the Database table name. This property is optional for all database types. If not provided, the table name will be automatically assigned as in 1 above.
		Note: When referring to excel ranges, do not include the header row in the range.

`sortIndexCols`
or
`sort`
	`"sortIndexCols": true`	Use this property to sort the columns alphabetically.

`Type`	`"type": "excel"`
	`"type": "odbc"`
	`"type": "csv"`
		Use this property to pass the file type: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "odata" (OData database), "mssql" (Microsoft Sequel), "oracle" (Oracle database), "csv" (Comma Separated Value), "json" (JSON file), "stage" or "xml" (XML file).

		The last two file types "json" and "xml", may be used for referencing stored models in external JSON and XML (PMML) files. The "json" file type may also be used for reading XLMiner::DataFrame serialized into JSON and stored in an external JSON file.
`valuecol`	`"valueCol": ["Initials"]` This property creates a RASON table and should only be used in conjunction with valueCols, not colIndex and rowIndex which create a dataframe.	Used with `binding` property to bind imported values from a readable data source. If omitted, the RASON interpreter assumes the last column in the table as the input to `valueCol`.
`valueCols`	`"valueCols": ["qty"]`	Used in conjunction with `indexCols`. Use this property to import columns/fields containing values

Datasets ("datasets")

You'll use this section to further refine data imported from within "datasources". The section, "datasets", is an object with user defined attributes where each attribute defines an object containing the following properties. See the example below.

```
"datasets": {
      "myData": {
            "binding": "mySrc",
            "targetCol":"Y"
    }
  },
```

Properties associated with "datasets" are listed in the table below.

Data Set Property	**Example**	**Explanation**
`binding`	`"binding":"mySrc"`	In the example above, this property binds imported data, `mySrc`, (from within `dataSource`) to a new dataset, "myData". (String property)
`colNames`	`"colNames": ["A", "B", "C", "D", "E", "F", "G"]`	Assigns column names. (Array property)
`dataCol`	`"dataCol":"CHAS"`	Selects input variables. (String property)
`excludedCols`	`"excludedCols": ["CHAS", "MEDV"]`	Excludes specified columns from the data mining method. (Array property)
`indexCols`	`indexCols: ["parts", "prods"]`	Used in conjunction with `valueCols.` Use this property to index by dimension(s).
`rowNames`	`"rowNames": ["record1", "record2"]`	Assigns row names. (Array property)

selectedCols	"selectedCols": ['X1', 'X2', 'X3', 'X4']	Selects specified columns. (Array property)
strataCol	"strataCol":"Y"	Selects the stratum variable when performing stratified random sampling. (String property)
targetCol	"targetCol":"Y"	Use this property to pass the name of the output column. (String property)
timeVariable	"timeVariable":"Year"	Selects the Time variable in a time series dataset. (String property)
value	"value": [["black", null, 6.0, 2.0, 1.0, "nan", 1], ["", 3.0, 9.0, 5.1, null, "", 2], ["red", 7.0, 8.0, null, 9.2, "small", 3], ["red", 10000.0, null, 4.4, 4.4, "large", -1], ["blue", 2, 3, 5.6, 3.4, "unknown", 5]],	Sets the values of the array. Sets the values of a table.
weights	"weights": [1.0,2.1,...]	Using a Weight variable allows the user to allocate a weight to each record. A record with a large weight will influence the model more than a record with a smaller weight. (Array property)

Weaklearner ("weakLearner")

This section is used (only) to specify a weak learner in a bagging or boosting estimator algorithm. Two examples are displayed below. The first example selects the decision tree algorithm to perform a classification and the second selects the decision tree algorithm to perform a regression.

```
weakLearner: {
    treeWeakLearner: {
      type: 'classification',
      algorithm: 'decisionTree',
      parameters: {
        minNumRecordsInLeaves: 2
      }
    }
  },

  weakLearner: {
    treeWeakLearner: {
      type: 'regression',
      algorithm: 'decisionTree',
      parameters: {
        minNumRecordsInLeaves: 2
      }
    }
  },
```

The following properties are available for use.

weakLearner Property	**Example**	**Explanation**

type	"type":"classification" "type":"regression"	Use this property to specify whether the estimator will be a classification or regression estimator.
algorithm	"algorithm":"decisionTree" **Parameter Options** • decisionTree • discriminantAnalysis • linearRegression • logisticRegression • naiveBayes • nearestNeighbors • neuralNetwork	Use this property to select the weak learner. See the Example column for a list of choices.

parameters	"parameters" : {
	"method": "M1_BREIMAN",
	"numWeakLearners": 2,
	"resamplingSeed": 10
	}
		Use this property to set parameter values or turn parameters on or off using "true" or "false". For a full list of parameters, see below.

Algorithm Parameters: Decision Tree

Parameters	Option Settings or Example	Explanation
priorProbMethod	"priorProbMethod":"EMPIRICAL" **Parameter Options** • EMPIRICAL • UNIFORM • MANUAL	**For classification only** Use EMPIRICAL when the probability of encountering a particular class in the dataset is the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability.
maxNumLevels	"maxNumLevels":3	This option specifies the maximum number of levels in the tree.
maxNumNodes	"maxNumNodes":5	This option specifies the maximum number of nodes in the tree.
minNumRecordsInLeaves	"minNumRecordsInLeaves":5	This option specifies the minimum , number of records allowed in terminal nodes, or leaves of the tree.
maxNumSplits	"maxNumSplits":10	This option specifies the maximum number of splits in the tree.

Algorithm Parameters: Discriminant Analysis

Parameters	Option Settings or Example	Explanation
priorProbMethod	"priorProbMethod":"EMPIRICAL"	Use EMPIRICAL when the probability of encountering a particular class in the dataset is

	`Parameter Options` - `EMPIRICAL` - `UNIFORM` - `MANUAL`	the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability.

`quadratic`	`"quadratic": true`
	`"quadratic": false`
		Use **True** to use **Quadratic Discriminant Analysis** (QDA). QDA produces a quadratic decision boundary.
		Use **False** to use **Line**ar **Discriminant Analysis** (LDA). LDA produces a linear decision boundary.
		Both QDA and LDA assume that the data is normally distributed and each class has it's own mean. However, while LDA assumes that the covariance matrix for each class is the *same*, QDA assumes that the covariance matrices for each class are *different*.
		QDA is a more flexible technique when compared to LDA. QDA's performance improves over LDA when the class covariance matrices are disparate. Since each class has a different covariance matrix, the number of parameters that must be estimated increases significantly as the number of dimensions (predictors) increase. As a result, LDA might be a better choice over QDA on datasets with small numbers of observations and large numbers of classes. It's advisable to try both techniques to determine which one performs best on your model. You can easily switch between LDA and QDA simply by setting this option to true or false.

Algorithm Parameters: Linear Regression

Parameters	Option Settings or Example	Explanation
`fitIntercept`	`"fitIntercept": true`	When this option is set to true, the default setting, the linear regression intercept will be fit. When this option is set to false, the intercept term will be forced to 0.

Algorithm Parameters: Logistic Regression

Parameters	Option Settings or Example	Explanation
`fitIntercept`	`"fitIntercept": true`	When this option is set to true, the default setting, the logistic regression intercept will be fit. When this option is set to false, the intercept term will be forced to 0.

`priorProbMethod`	`"priorProbMethod":"EMPIRICAL"` **Parameter Options** • `EMPIRICAL` • `UNIFORM` • `MANUAL`	For classification only Use EMPIRICAL when the probability of encountering a particular class in the dataset is the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability.
`maxIterations`	`"maxIterations": 30`	Estimating the coefficients in the Logistic Regression algorithm requires an iterative non-linear maximization procedure. You can specify a maximum number of iterations to prevent the program from getting lost in very lengthy iterative loops. This value must be an integer greater than 0 or less than or equal to 200 (0 < value <= 200).

Algorithm Parameters: Naïve Bayes

Parameters	Option Settings or Example	Explanation
`laplaceSmoothing`	`"laplaceSmoothing": true`	If a particular realization of some feature never occurs in a given class in the training partition, then the corresponding frequency-based prior conditional probability estimate will be zero. To mitigate this problem, this parameter allows users to specify a small correction value, known as a pseudocount, so that no probability estimate is ever set to 0. A Pseudocount set to zero is equivalent to no smoothing. Any positive value for this parameter is accepted.
`priorProbMethod`	`"priorProbMethod":"EMPIRICAL"` **Parameter Options** • `EMPIRICAL` • `UNIFORM` • `MANUAL`	**For classification only** Use EMPIRICAL when the probability of encountering a particular class in the dataset is the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability.
`smoothingAlpha`	`"smoothingAlpha":1`	Setting this option to zero is equivalent to no smoothing.

Algorithm Parameters: Nearest Neighbors

Parameters	Option Settings or Example	Explanation
`priorProbMethod`	`"priorProbMethod":"EMPIRICAL"`	**For classification only**

	Parameter Options - EMPIRICAL - UNIFORM - MANUAL	Use EMPIRICAL when the probability of encountering a particular class in the dataset is the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability.
`numNeighbors`	`"numNeighbors":3`	This is the parameter k in the k-Nearest Neighbor algorithm.

Algorithm Parameters: Neural Network

Parameters	Option Settings or Example	Explanation
`dataForErrorComputation`	`"dataForErrorComputation" : "TRAIN_AND_VALID"` **Parameter Options** - ONLY_TRAIN - ONLY_VALID - TRAIN_AND_VALID	Specifies the data partition to be used to estimate the error after each training epoch.
`errorTolerance`	`"errorTolerance": 0.01`	Use this option to set the error tolerance. The error in a particular iteration is backpropagated only if it is greater than the value specified for this option. Typically error tolerance is a small value in the range from 0 to 1.
`hiddenLayerActivation`	`"hiddenLayerActivation": "LOGISTIC_SIGMOID"` **Parameter Options** - LOGISTIC_SIGMOID - SOFTMAX - TANH	Nodes in the hidden layer receive input from the input layer. The output of the hidden nodes is a weighted sum of the input values. This weighted sum is computed with weights that are initially set at random values. As the network "learns", these weights are adjusted. This weighted sum is used to compute the hidden node's output using a transfer function. Select Sigmoid (the default setting) to use a logistic function for the transfer function with a range of 0 and 1. This function has a "squashing effect" on very small or very large values but is almost linear in the range where the value of the function is between 0.1 and 0.9. Select Hyperbolic Tangent to use the tanh function for the transfer function, the range being -1 to 1. If more than one hidden layer exists, this function is used for all layers.
`learningOrder`	`"learningOrder": "RANDOM"` **Parameter Options** - ORIGINAL - RANDOM	This option sets the order in which the records in the training dataset are processed. It is recommended to shuffle the training data to avoid the possibility of processing correlated records in order which can help the neural network algorithm to converge faster. Use

		RANDOM to randomly shuffle the data. ORIGINAL uses the original order of records.
learningOrderSeed	"learningOrderSeed":12345	This option specifies the seed for shuffling the training records. Note that different random shuffling may lead to different results, but as long as the training data is shuffled, different ordering typically does not result in drastic changes in performance.
learningRate	"learningRate":0.4	This option sets the multiplying factor for the error correction during backpropagation; it is roughly equivalent to the learning rate for the neural network. A low value produces slow but steady learning, a high value produces rapid but erratic learning. Values for the step size typically range from 0.000001 to 1.0.
maxNumEpochsWithNoImprovement	"maxNumEpochsWithNoImprovement": 30	The algorithm will stop after this number of epochs has been completed and no improvement has been realized.
maxTrainingTimeSeconds	"maxTrainingTimeSeconds":5.0	The algorithm will stop once this time (in seconds) has been exceeded.
minRelativeErrorChangeComparedToNullModel	"minRelativeErrorChangeComparedToNullModel":0.0001	If the relative change in error compared to the Null Model is less than this value, the algorithm will stop. Null Model is the baseline model used for comparing the performance of the neural network model.
minRelativeErrorChange	"minRelativeErrorChange": 0.00001,	If the relative change in error is less than this value, the algorithm will stop.
numEpochs	"numEpochs":100	Use this option to set the number of epochs, or one sweep through all records in the training set.
numNeurons	"numNeurons":[5,4]	Use this open to specify the number of neurons in the Neural Network Architecture i.e. the number of neurons in the hidden layer(s). The first value will determine the number of neurons in the 1st hidden layer, the second value determines the number of neurons in the 2nd layer, and so on.
outputLayerActivation	"outputLayerActivation": "TANH" **Parameter Options** • LOGISTIC_SIGMOID • SOFTMAX • TANH	The output layer is also computed using the same transfer function as described for setHiddenLayerActivation. Select Sigmoid (the default setting) to use a logistic function for the transfer function with a range of 0 and 1. Select Hyperbolic Tangent to use the tanh function for the transfer function, the range being -1 to 1. In neural networks, the Softmax function is often implemented at the final layer of a classification neural network to impose the constraints that the posterior

		probabilities for the output variable must be >= 0 and <= 1 and sum to 1.
priorProbMethod	"priorProbMethod": "EMPIRICAL" **Parameter Options** • EMPIRICAL • MANUAL • UNIFORM	**For classification only** Use EMPIRICAL when the probability of encountering a particular class in the dataset is the same as the frequency with which it occurs in the training data. Use MANUAL to enter the desired class and probability. Use UNIFORM when all classes occur with equal probability.
responseCorrection	"responseCorrection":0.01	This option specifies the value applied to the Normalization rescaling formula, if the output layer activation is Sigmoid (or Softmax in Classification) or Adjusted Normalization, if the output layer activation is Hyperbolic Tangent. The rescaling correction ensures that all response values stay within the range of activation function.
weightDecay	"weightDecay":0.0	Use this option to prevent over-fitting of the network on the training data. Set a weight decay to penalize the weight in each iteration. Each calculated weight will be multiplied by (1-decay).
weightInitSeed	"weightInitSeed":12345	Use this option to initialize the Random Seed for Weights Initialization. This value is used to set the seed for the initial assignment of the neuron values. Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the same sequence of random numbers is used each time the neuron values are calculated. The default value is “12345”.
weightMomentum	"weightMomentum":0.0	This option controls the weight change momentum in the neural network algorithm. In each new round of error correction, some memory of the prior correction is retained so that an emerging outlier does not spoil the accumulated learning.

Estimator ("estimator")

This section is where the "estimator" is defined and is applicable only to algorithms that "fit" a model. In other words, an "estimator" estimates "something" from the training data and stores it in a fitted model to be used later. Every algorithm that implements a "fit()" interface has an Estimator – Model pair. This section is mutually exclusive with the "transformer" section. **Both may not appear in the same Rason model.** Data mining algorithms that "fit" a model include: Rescaler, Principal Components Analysis, Binning, Factorization, Encoding, Canonical Variate Analysis, Imputation, k-Means Clustering, Hierarchical Clustering, TF-IDF (Text Mining), Latent Semantic Analysis (Text Mining), ARIMA, all Smoothing methods (Exponential, Double

Exponential, Moving Average, and Holt Winters) and all classification and regression methods (Decision Trees, Ensemble Methods, Discriminant, Linear Regression, Logistic Regression, Naïve Bayes, Nearest Neighbors, and Neural Networks).

In the example below, the estimator "mlrEstimator" uses the linear regression algorithm to fit a model.

```
"estimator": {
    "mlrEstimator": {
      "type": "regression",
      "algorithm": "linearRegression",
      "parameters": {
        "fitIntercept": true
      }
    }
  },
```

This section includes the following properties.

Property	Example	Definition

`type`	`"type":"regression"`
	
	Parameter Options
	`"classification"`
	`"clustering"`
	`"regression"`
	`"textMining"`
	`"timeSeries"`
	`"transform"`
		Use this property to specify the type of algorithm to be applied: classification, regression, clustering, text mining, transformation or time series.

`algorithm`	`"algorithm": "boosting"`	Available option settings will vary depending on the type setting. Use this property to specify the algorithm to be used to "fit" the model. See the chart below for all options.

`parameters`	`"parameters" : {`
	`"method": "M1_BREIMAN",`
	`"numWeakLearners": 2,`
	`"resamplingSeed": 10`
	`}`
		Available parameter settings will vary depending on the algorithm setting. Use this property to set parameter values or turn parameters on or off using "true" or "false". For a full list of parameters, see below.

The chart below contains the available options for "algorithm" based on the "type" argument.

If "type" =	Algorithm option settings	Definition
`"classification"`	`"algorithm" : "boosting"` **Parameter Options** • `bagging` • `boosting` • `decisionTree` • `findBestModel` • `DiscriminantAnalysis`	boosting – Runs the Boosting ensemble method for classification. bagging – Runs the Bagging ensemble method for classification. decisionTree – Runs the Decision Tree algorithm for classification.

	• `logisticRegression` • `naïveBayes` • `nearestNeighbors` • `neuralNetwork` • `randomTrees`	findBestModel – Runs the Find Best Model method. For more details, see the Data Mining chapter within the RASON User Guide. DiscriminantAnalysis – Runs the Discriminant Analysis classification algorithm. logisticRegression – Runs the Logistic Regression classification algorithm. naïveBayes – Runs the Naïve Bayes classification algorithm. nearestNeighbors – Runs the k-Nearest Neighbors algorithm for classification. neuralNetwork – Runs the Neural Network algorithm for classification. randomTrees – Runs the Random Trees ensemble method for classification.
`"clustering"`	`"algorithm" : "hierarchical"` **Parameter Options** • `hierarchical` • `kMeans`	kmeans – Performs clustering using the k Means Clustering algorithm. hierarchical – Performs clustering using the Hierarchical Clustering algorithm.
`"regression"`	`"algorithm" : "boosting"` **Parameter Options** • `bagging` • `boosting` • `decisionTree` • `findBestModel` • `linearRegression` • `nearestNeighbors` • `neuralNetwork` • `randomTrees` • `findBestModel`	boosting – Runs the Boosting ensemble method for regression. bagging – Runs the Bagging ensemble method for regression. decisionTree – Runs the Decision Tree algorithm for regression. findBestModel – Runs the Find Best Model method. For more details, see the Data Mining chapter within the RASON User Guide. linearRegression – Runs the Linear Regression regression algorithm. nearestNeighbors – Runs the k-Nearest Neighbors algorithm for regression. neuralNetwork – Runs the Neural Network algorithm for regression. randomTrees – Runs the Random Trees ensemble method for regression.
`"textMining"`	`"algorithm" : "tfIdf"` **Parameter Options** • `latentSemanticAnalysis` • `tfIdf`	tfIdf – Performs text mining using Term Frequency – Inverse Document Frequency Vectorization (TF-IDF). latentSemanticAnalysis – Performs text mining using Latent Semantic Analysis (LSA).
`"transform"`	`"algorithm" : "binning"`	binning – Use to group measured data into data classes.

	Parameter Options • binning • canonicalVariateAnalysis • factorization • imputation • linearWrapping • logisticWrapping • oneHotEncoding • principalComponentAnalysis • rescaling • univariate	canonicalVariateAnalysis – Produces the Canonical Variates. factorization – Use to create category scores. imputation – Use to handle missing values. linearWrapping – Performs Feature Selection (on a continuous output variable) using linear wrapping. logisticWrapping **-** Performs Feature Selection (on a categorical output variable) using logistic wrapping. oneHotEncoding – Use to create dummy variables. principalComponentAnalysis – Use to run PCA. rescaling – Use to rescale data univariate – Performs Feature Analysis by ranking variables according to one or more univariate measures.
"timeSeries"	"algorithm" : "addHoltWinters" **Parameter Options** • addHoltWinters • arima • doubleExponential • exponential • lagAnalysis • movingAverage • mulHoltWinters • noTrendHoltWinters	addHoltWinters – Runs the Additive Holt Winters Smoothing method. arima – Performs Time Series Analysis using ARIMA. doubleExponential – Runs the Double Exponential Smoothing method. exponential – Runs the Exponential Smoothing method. lagAnalysis **--** Performs Time Series Analysis using lag analysis. movingAverage – Runs the Moving Average smoothing method. mulHoltWinters **-** Runs the Multiplicative Holt Winters Smoothing method. noTrendWintersHoltWinters **-** Runs the No Trend Holt Winters Smoothing method.

The chart below contains the available options for "parameters" based on the "algorithm" argument.

Algorithm Parameters Common to All Classification Algorithms

Parameters	Option Settings or Example	Explanation

priorProb	"priorProb": [
	   ['1', 0.4],
	   ['0', 0.6]
	],
		For classification models only.
		Specifies the desired class and probability values

priorProbMethod	"priorProbMethod":"EMPIRICAL" **Parameter Options**	**For classification models only.** Use EMPIRICAL when the probability of encountering a particular class in the dataset is

	• `EMPIRICAL` • `UNIFORM` • `MANUAL`	the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability. See the example to the left.

Algorithm Parameters: Find Best Model for Classification & Regression

Parameters	Option Settings or Example	Explanation

`learners`	`Full Specification`
	`"estimator": {`
	`  "fbmEstimator": {`
	`    "type": "regression",`
	`    "algorithm":`
	`     "findBestModel",`
	`    "learners": {`
	`      "linearRegression": {`
	`        "fitIntercept": false`
	`      },`
	`      "nearestNeighbors": {`
	`        "numNeighbors": 5`
	`      },`
	`      "neuralNetwork": {}`
	`    }`
	`  }`
	`}`
	
	`Partial Specification`
	`"estimator": {`
	`  "fbmEstimator": {`
	`    "type": "regression",`
	`    "algorithm":`
	`     "findBestModel",`
	`    "learners": [`
	`       "linearRegression",`
	`       "nearestNeighbors",`
	`       "neuralNetwork"`
	`    ]`
	`  }`
	`}`
	
	`Default Specification`
	`"estimator": {`
	`  "fbmEstimator": {`
	`    "type": "regression",`
	`    "algorithm":`
	`     "findBestModel",`
	`  }`
	`}`
		Full Specification: User specifies all available learners and may edit parameter settings. In the Full Specification example to the left, three learners will be available to the Find Best Model method: linearRegression, nearestNeighbors and neuralNetwork. Parameter settings are defined for just two learners: linearRegression and nearestNeighbors. The nueralNetwork learner will use it's default parameter settings. Refer to each learner's chart in this section for a list of all available parameters.
		Partial Specification: User specifies all available learners. Default parameter settings will be used. In the Partial Specification example to the left, three learners will be available to the Find Best Model method: linearRegression, nearestNeighbors and neuralNetwork.
		When using the **Default Specification**, no learners are specified. All learners will be available to the Find Best Model method.
		
		Learners available to classification models are: bagging, boosting, decisionTree, discriminantAnalysis, logisticRegression, nearestNeighbors, neuralNetwork and randomTrees.
		
		Learners available to regression models are: bagging, boosting, decisionTree, linearRegression, nearestNeighbors, neuralNetwork and randomTrees.

Algorithm Parameters: Discriminant Analysis for Classification

Parameters	Option Settings or Example	Explanation

`quadratic`	`"quadratic": true`
	`"quadratic": false`
		Use **True** to use **Quadratic Discriminant Analysis** (QDA). QDA produces a quadratic decision boundary.
		Use **False** to use **Linear Discriminant Analysis** (LDA). LDA produces a linear decision boundary.
		Both QDA and LDA assume that the data is normally distributed and each class has it's own mean. However, while LDA assumes that the covariance matrix for each class is the *same*, QDA assumes that the covariance matrices for each class are *different*.
		QDA is a more flexible technique when compared to LDA. QDA's performance improves over LDA when the class covariance matrices are disparate. Since each class has a different covariance matrix, the number of parameters that must be estimated increases significantly as the number of dimensions (predictors) increase. As a result, LDA might be a better choice over QDA on datasets with small numbers of observations and large numbers of classes. It's advisable to try both techniques to determine which one performs best on your model. You can easily switch between LDA and QDA simply by setting this option to true or false.

Algorithm Parameters: Logistic Regression for Classification

Parameters	Option Settings or Example	Explanation
`categoricalFeaturesNames`	`"categoricalFeaturesNames": [ "X1" ]`	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.
`confidenceLevel`	`"confidenceLevel":0.95`	Sets the confidence level for the Coefficients Confidence Interval.
`fitIntercept`	`"fitIntercept":false`	Set "fitIntercept":false to force the intercept term to 0. Set this option to "true", the default, to fit the Logistic Regression intercept.
`maxIterations`	`"maxIterations": 30`	Estimating the coefficients in the Logistic Regression algorithm requires an iterative non-linear maximization procedure. You can specify a maximum number of iterations to

		prevent the program from getting lost in very lengthy iterative loops. This value must be an integer greater than 0 or less than or equal to 200 (1 < value <= 200).
`successClass`	`"successClass":"1"`	Select the success value for the output variable here (i.e. 0 or 1 or “yes” or “no”).
`weights`	`"weights": [1.0,2.1,...]`	Provides a weight variable allowing the user to allocate a weight to each record. A record with a large weight will influence the model more than a record with a smaller weight.

Algorithm Parameters: Naïve Bayes for Classification

Parameters	Option Settings or Example	Explanation
`laplaceSmoothing`	`"laplaceSmoothing": true`	Set this option to True to turn on LaPlace Smoothing. Set the Pseudocount using the option, smoothingAlpha.
`smoothingAlpha`	`"smoothingAlpha":1`	Use this option to specify a small correction value, known as a pseudocount, so that no probability estimate is ever set to 0. A pseudocount set to zero is equivalent to no smoothing. When Laplace Smoothing is turned on, any positive value for psuedocount will be accepted.

Algorithm: Bagging for Classification or Regression

Parameters	Example/Parameter Options	Definition
`bootstrapSeed`	`"bootstrapSeed": 10`	This value specifies the seed for random resampling of the training data for each weak learner. Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the same sequence of random numbers is used each time the dataset is chosen for the classifier. The default value is "12345".
`numWeakLearners`	`"numWeakLearners" : 4`	This option controls the number of “weak” classification/regression models that will be created. The ensemble method will stop when the number of models created reaches the value set for this option. The algorithm will then compute the weighted sum of votes for each class and assign the “winning” classification to each record.

Algorithm: Boosting for Classification or Regression

Parameter	Example/Parameter Options	Definition

`method`	`"method" : "M1_BREIMAN"`
	`Parameter Options`
		For Classification models only.

	• `M1_BREIMAN` • `M1_FREUND` • `SAMME`	The difference in the algorithms is the way in which the weights assigned to each observation or record are updated. In AdaBoost.M1 (Freund), the constant is calculated as: αb= ln((1-eb)/eb) In AdaBoost.M1 (Breiman), the constant is calculated as: αb= 1/2ln((1-eb)/eb) In SAMME, the constant is calculated as: αb= 1/2ln((1-eb)/eb + ln(k-1) where k is the number of classes (When the number of categories is equal to 2, SAMME behaves the same as AdaBoost Breiman.)
`numWeakLearners`	`"numWeakLearners" : 4`	This option controls the number of "weak" classification/regression models that will be created. The ensemble method will stop when the number of models created reaches the value set for this option. The algorithm will then compute the weighted sum of votes for each class and assign the "winning" classification to each record.
`resamplingSeed`	`"resamplingSeed": 10`	**For Classification models only.** This value specifies the seed for random resampling of the training data for each weak learner. Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the same sequence of random numbers is used each time the dataset is chosen for the classifier. The default value is "12345".
`stepSize`	`"stepSize": 0.3`	**For Regression models only.** The Adaboost algorithm minimizes a loss function using the gradient descent method. The Step size parameter is used to ensure that the algorithm does not descend too far when moving to the next step. It is recommended to leave this option at the default of 0.3, but any number between 0 and 1 is acceptable. A Step size setting closer to 0 results in the algorithm taking smaller steps to the next point, while a setting closer to 1 results in the algorithm taking larger steps towards the next point.

Algorithm Parameters: Neural Network for Classification or Regression

Parameters	Option Settings or Example	Explanation
`categoricalFeaturesNames`	`"categoricalFeaturesNames": [ 'X1' ]`	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.

costFunction	"costFunction": "SUM_OF_SQUARES" **Parameter Options for Classification** • SUM_OF_SQUARES • CROSS_ENTROPY • LOG_LOSS **Parameter Options for Regression** • SUM_OF_SQUARES	Sets the cost function. The cost function measures "how well" a neural network performed with respect to a given training dataset and the expected output. If not set, this option will be set automatically based on the classification/regression and output layer activation.
dataForErrorComputation	"dataForErrorComputation" : "TRAIN_AND_VALID" **Parameter Options** • ONLY_TRAIN • ONLY_VALID • TRAIN_AND_VALID	Specifies the data partition to be used to estimate the error after each training epoch.
errorTolerance	"errorTolerance": 0.01	Use this option to set the error tolerance. The error in a particular iteration is backpropagated only if it is greater than the value specified for this option. Typically error tolerance is a small value in the range from 0 to 1.
hiddenLayerActivation	"hiddenLayerActivation":"LOGISTIC_SIGMOID" **Parameter Options** • LOGISTIC_SIGMOID • SOFTMAX • TANH	Nodes in the hidden layer receive input from the input layer. The output of the hidden nodes is a weighted sum of the input values. This weighted sum is computed with weights that are initially set at random values. As the network “learns”, these weights are adjusted. This weighted sum is used to compute the hidden node’s output using a transfer function. Select Sigmoid (the default setting) to use a logistic function for the transfer function with a range of 0 and 1. This function has a “squashing effect” on very small or very large values but is almost linear in the range where the value of the function is between 0.1 and 0.9. Select Hyperbolic Tangent to use the tanh function for the transfer function, the range being -1 to 1. If more than one hidden layer exists, this function is used for all layers.
learningOrder	"learningOrder":"RANDOM"	This option sets the order in which the records in the training dataset are processed. It is recommended to shuffle the training data to avoid the possibility of processing correlated records in order which can help the neural network algorithm to converge faster. Use RANDOM to randomly shuffle the data. If ORIGINAL, the original order of records will be used.
learningOrderSeed	"learningOrderSeed":12345	This option specifies the seed for shuffling the training records. Note that different random

		shuffling may lead to different results, but as long as the training data is shuffled, different ordering typically does not result in drastic changes in performance.
`learningRate`	`"learningRate":0.4`	This option sets the multiplying factor for the error correction during backpropagation; it is roughly equivalent to the learning rate for the neural network. A low value produces slow but steady learning, a high value produces rapid but erratic learning. Values for the step size typically range from 0.000001 to 1.0.
`maxNumEpochsWithNoImprovement`	`"maxNumEpochsWithNoImprovement":30`	The algorithm will stop after this number of epochs has been completed and no improvement has been realized.
`maxTrainingTimeSeconds`	`"maxTrainingTimeSeconds":5.0`	The algorithm will stop once this time (in seconds) has been exceeded.
`minRelativeErrorChangeComparedToNullModel`	`"minRelativeErrorChangeComparedToNullModel":0.0001`	If the relative change in error compared to the Null Model is less than this value, the algorithm will stop. Null Model is the baseline model used for comparing the performance of the neural network model.
`minRelativeErrorChange`	`"minRelativeErrorChange":0.00001`	If the relative change in error is less than this value, the algorithm will stop.
`numEpochs`	`"numEpochs":100`	Use this option to set the number of epochs, or one sweep through all records in the training set.
`numNeurons`	`"numNeurons":[5,4]`	Use this open to specify the number of neurons in the Neural Network Architecture i.e. the number of neurons in the hidden layer(s). The first value will determine the number of neurons in the 1st hidden layer, the second value determines the number of neurons in the 2nd layer, and so on.
`outputLayerActivation`	`"outputLayerActivation":"TANH"` **Parameter Options** • `LOGISTIC_SIGMOID` • `SOFTMAX` • `TANH`	The output layer is also computed using the same transfer function as described for `setHiddenLayerActivation`. Select Sigmoid (the default setting) to use a logistic function for the transfer function with a range of 0 and 1. Select Hyperbolic Tangent to use the tanh function for the transfer function, the range being -1 to 1. In neural networks, the Softmax function is often implemented at the final layer of a classification neural network to impose the constraints that the posterior probabilities for the output variable must be >= 0 and <= 1 and sum to 1.

weightDecay	"weightDecay":0.0	Use this option to prevent over-fitting of the network on the training data. Set a weight decay to penalize the weight in each iteration. Each calculated weight will be multiplied by (1-decay).
weightInitSeed	"weightInitSeed":12345	Use this option to initialize the Random Seed for Weights Initialization. This value is used to set the seed for the initial assignment of the neuron values. Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the same sequence of random numbers is used each time the neuron values are calculated. The default value is "12345".
weightMomentum	"weightMomentum":0.0	This option controls the weight change momentum in the neural network algorithm. In each new round of error correction, some memory of the prior correction is retained so that an emerging outlier does not spoil the accumulated learning.

Algorithm Parameters: Decision Trees for Classification or Regression

Parameters	Option Settings or Example	Explanation
categoricalFeaturesNames	"categoricalFeaturesNames": ["X1"]	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.
prunedTreeType	"prunedTreeType": "MIN_ERROR" **Parameter Options** • FULL_GROWN • BEST_PRUNED • MIN_ERROR • MANUAL	Use this option to select the tree used to score the validation dataset: FULL_GROWN, BEST_PRUNED, MIN_ERROR or MANUAL. • Use "FULL_GROWN" to use the complete tree. • Use "BEST_PRUNED" to use the Best Pruned Tree. • Use "MIN_ERROR" to use the Minimum Error Tree. • Use "MANUAL" to create a tree with a specified number of decision nodes. When this option is used, set the number of decision nodes in the Pruned Tree using "prunedTreeNumDecisionNodes"
maxNumLevels	"maxNumLevels":3	This option specifies the maximum number of levels in the tree.
maxNumLevelsTreeDiagram	"maxNumLevelsTreeDiagram": 7	This option specifies the maximum number of levels in the tree to be included in the output.

maxNumNodes	"maxNumNodes":5	This option specifies the maximum number of nodes in the tree.
maxNumSplits	"maxNumSplits":10	This option specifies the maximum number of splits in the tree.
minNumRecordsInLeaves	"minNumRecordsInLeaves":5	This option specifies the minimum number of records allowed in terminal nodes, or leaves of the tree.
prunedTreeNumDecisionNodes	"prunedTreeNumDecisionNodes":5	Used in conjuction with "prunedTree Type" : "MANUAL". Use this option to specify the number of decision nodes in the pruned tree.
weights	"weights": [1.0,2.1,...]	Provides a weight variable allowing the user to allocate a weight to each record. A record with a large weight will influence the model more than a record with a smaller weight.

Algorithm Parameters: Random Trees for Classification or Regression

Parameters	Option Settings or Example	Explanation
bootstrapSeed	"bootstrapSeed": 1	This value sets the bootstrapping random number seed. Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the same sequence of random numbers is used each time the dataset is chosen for the classifier. The default value is “12345".
categoricalFeaturesNames	"categoricalFeaturesNames": ["X1"]	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.
featureSelectionSeed	"featureSelectionSeed":2	This value sets the feature selection random number seed. Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the same sequence of random numbers is used each time the dataset is chosen for the classifier. The default value is “12345”.
maxNumLevels	"maxNumLevels":3	This option specifies the maximum number of levels in the tree.
maxNumNodes	"maxNumNodes":5	This option specifies the maximum number of nodes in the tree.
maxNumSplits	"maxNumSplits":10	This option specifies the maximum number of splits in the tree.
minNumRecordsInLeaves	"minNumRecordsInLeaves":5	This option specifies the minimum number of records allowed in terminal nodes, or leaves of the tree.
numSelectedFeatures	"numSelectedFeatures": 5	The Random Trees ensemble method works by training multiple “weak” classification

		trees using a fixed number of randomly selected features then taking the mode of each class to create a “strong” classifier. This option controls the fixed number of randomly selected features in the algorithm. The default setting is 3.
`numWeakLearners`	`"numWeakLearners" : 4`	This option controls the number of “weak” classification/regression models that will be created. The ensemble method will stop when the number of models created reaches the value set for this option. The algorithm will then compute the weighted sum of votes for each class and assign the “winning” classification to each record.

Algorithm Parameters: k-Nearest Neighbors for Classification or Regression

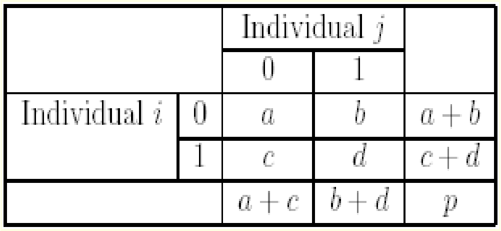
Parameters	Option Settings or Example	Explanation
`includeTies`	`"includeTies": true`	If includeTies = True, all points with distance equal to kth nearest neighbor are included in the result. If includeTies = False, exactly k nearest neighbors are returned.
`numNeighbors`	`"numNeighbors":3`	This is the parameter k in the k-nearest neighbor algorithm.
`stable`	`"stable":true`	If stable = true, the tied neighbors (up to kth neighbor) remain in the original order. If stable = false, the tied neighbors (up to kth neighbor) are in pseudo-random order.

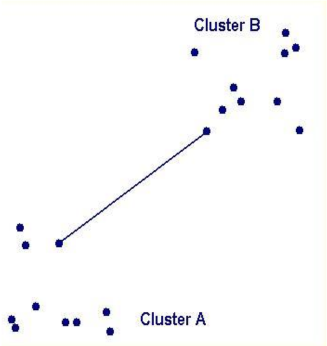
`weightingScheme`	`"weightingScheme":`
	`"INVERSE_DISTANCE"`
	Parameter Options
	• `EQUAL`
	• `INVERSE_DISTANCE`
		Use this option to select the weighting scheme: equal or inverse distance.

`weights`	`"weights": [1.0,2.1,...]`	Provides a weight variable allowing the user to allocate a weight to each record. A record with a large weight will influence the model more than a record with a smaller weight.

Algorithm Parameters: Hierarchical for Clustering

Parameters	Option Settings or Example	Explanation
`dissimilarity`	`"dissimilarity" : "EUCLIDEAN"`	EUCLIDEAN -- Hierarchical clustering uses the Euclidean Distance as the similarity measure for working on raw numeric data.

<table>
<tr>
<td></td>
<td></td>
<td>
<p>When the data is binary, the remaining two options, Jaccard's coefficients and Matching coefficient are available.</p>
<p>Suppose we have binary values for all the x_{ij} 's. See the table below for individual <i>i's</i> and <i>j's.</i></p>

<table>
<tr><td colspan="2"></td><td colspan="2">Individual j</td><td></td></tr>
<tr><td colspan="2"></td><td>0</td><td>1</td><td></td></tr>
<tr><td rowspan="2">Individual i</td><td>0</td><td>a</td><td>b</td><td>$a+b$</td></tr>
<tr><td>1</td><td>c</td><td>d</td><td>$c+d$</td></tr>
<tr><td colspan="2"></td><td>$a+c$</td><td>$b+d$</td><td>p</td></tr>
</table>
<p>The most useful similarity measures in this situation are:</p>
<p>JACCARD -- Jaccard's coefficient = d/(b+c+d). This coefficient ignores zero matches.</p>
<p>MATCHING -- The matching coefficient = (a + d)/p.</p>
</td>
</tr>
<tr>
<td><code>linkage</code></td>
<td>
<p><code>"linkage" : "SINGLE_LINKAGE"</code></p>
<p><b><code>Parameter Options</code></b></p>
<ul>
<li><code>CENTROID</code></li>
<li><code>COMPLETE_LINKAGE</code></li>
<li><code>GROUP_AVERAGE</code></li>
<li><code>MCQUITTY</code></li>
<li><code>MEDIAN</code></li>
<li><code>SINGLE_LINKAGE</code></li>
<li><code>WARD</code></li>
</ul>
</td>
<td>
<p>SINGLE_LINKAGE -- One of the simplest agglomerative hierarchical clustering methods is single linkage, also known as the nearest neighbor technique. The defining feature of this method is that distance between groups is defined as the distance between the closest pair of objects, where only pairs consisting of one object from each group are considered.</p>
<p>In this method, <b><i>D(r,s)</i></b> is computed as</p>
<p><b><i>D(r,s)</i></b> = Min { <i>d(i,j)</i> : Where object <i>i</i> is in cluster <b><i>r</i></b> and object <i>j</i> is cluster <b><i>s</i></b> }</p>
<p>Here the distance between every possible object pair <i>(i,j)</i> is computed, where object <i>i</i> is in cluster <b><i>r</i></b> and object <i>j</i> is in cluster <b><i>s</i></b>. The minimum value of these distances is said to be the distance between clusters <b><i>r</i></b> and <b><i>s.</i></b> In other words, the distance between two clusters is given by the value of the shortest link between the clusters.</p>
<p>At each stage of hierarchical clustering, the clusters <b><i>r</i></b> and <b><i>s</i></b>, for which <b><i>D(r,s)</i></b> is minimum, are merged.</p>
<p>This measure of inter-group distance is illustrated in the figure below:</p>
</td>
</tr>
</table>



COMPLETE_LINKAGE – This method, also called farthest neighbor clustering method, is the opposite of single linkage. In this clustering method, the distance between groups is defined as the distance between the most distant pair of objects, one from each group.

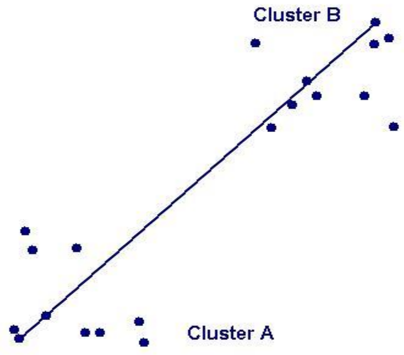
In the complete linkage method, ***D(r,s)*** is computed as

D(r,s) = Max { *d(i,j)* : Where object *i* is in cluster ***r*** and object *j* is cluster ***s*** }

Here the distance between every possible object pair *(i,j)* is computed, where object *i* is in cluster ***r*** and object *j* is in cluster ***s*** and the maximum value of these distances is said to be the distance between clusters ***r*** and ***s.*** In other words, the distance between two clusters is given by the value of the longest link between the clusters.

At each stage of hierarchical clustering, the clusters ***r*** and ***s***, for which ***D(r,s)*** is minimum, are merged.

The measure is illustrated in the figure below:



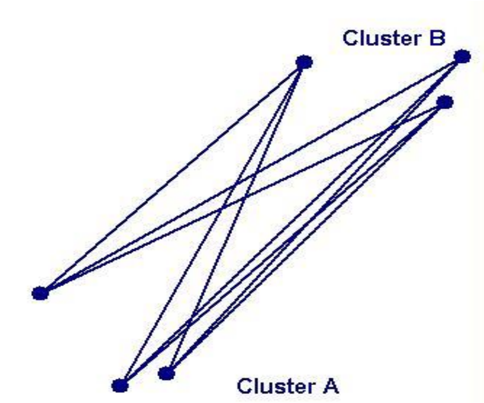
GROUP_AVERAGE **--** Here the distance between two clusters is defined as the average of distances between all pairs of objects, where each pair is made up of one object from each group.

In the average linkage method, ***D(r,s)*** is computed as

$$D(r,s) = T_{rs} / (N_r * N_s)$$

Where T_{rs} is the sum of all pairwise distances between cluster ***r*** and cluster ***s***. N_r and N_s are the sizes of the clusters ***r*** and ***s*** respectively**.**

At each stage of hierarchical clustering, the clusters ***r*** and ***s***, for which ***D(r,s)*** is the minimum, are merged. The figure below illustrates average linkage clustering:



CENTROID **--** With this method, groups once formed are represented by their mean values for each variable, that is, their mean vector, and inter-group distance is now defined in terms of distance between two such mean vectors.

In the group average linkage method, the two clusters **r** and **s** are merged such that, after merging, the average pairwise distance within the newly formed cluster, is minimized. Suppose we label the new cluster formed by merging clusters **r** and **s**, as **t**. Then **D(r,s)** , the distance between clusters **r** and **s** is computed as

D(r,s) = Average { d(i,j) : Where observations i and j are in cluster **t**, the cluster formed by merging clusters **r** and **s** }

At each stage of hierarchical clustering, the clusters **r** and **s**, for which **D(r,s)** is minimized, are merged. In this case, those two clusters are merged such that the newly formed cluster, on average, will have minimum pairwise distances between the points.

WARD **--** Ward (1963) proposed a clustering procedure seeking to form the partitions $P_{n,}$ $P_{n-1,........,}$ P_1 in a manner that minimizes the loss associated with each grouping, and to quantify

		that loss in a form that is readily interpretable. At each step in the analysis, the union of every possible cluster pair is considered and the two clusters whose fusion results in the minimum increase in 'information loss' are combined. Information loss is defined by Ward in terms of an error sum-of-squares criterion, ESS. The rationale behind Ward's proposal can be illustrated most simply by considering univariate data. Suppose for example, 10 objects have scores (2, 6, 5, 6, 2, 2, 2, 2, 0, 0, 0) on some particular variable. The loss of information that would result from treating the ten scores as one group with a mean of 2.5 is represented by ESS given by, $ESS_{\text{One group}} = (2 - 2.5)^2 + (6 - 2.5)^2 + ... + (0 - 2.5)^2 = 50.5$ On the other hand, if the 10 objects are classified according to their scores into four sets, {0,0,0}, {2,2,2,2}, {5}, {6,6} The ESS can be evaluated as the sum of squares of four separate error sums of squares $ESS_{\text{One group}} = ESS_{\text{group1}} + ESS_{\text{group2}} + ESS_{\text{group3}} + ESS_{\text{group4}} = 0.0$ Clustering the 10 scores into 4 clusters results in no loss of information. MCQUITTY -- When this procedure is selected, at each step, when two clusters are to be joined, the distance of the new cluster to an existing cluster is computed as the average of the distances from the proposed cluster to the existing cluster. MEDIAN -- The Median Method also uses averaging when calculating the distance between two records or observations. However, this method uses the median instead of the mean. One of the reasons why Hierarchical Clustering is so attractive to statisticians is that all possible clusters can be examined visually, or in any desired way, by examining the full dendrogram. However, there are a few limitations. 1. Hierarchical clustering can be computationally expensive as this method requires computing and storing an n x n distance matrix. If using a large dataset, this requirement can be very slow and require large amounts of memory.

		2. Clusters created through Hierarchical clustering are not very stable. If records are eliminated, the results can be very different. 3. Outliers in the data can impact the results negatively.
inputDataType	"inputDataType" : "DISTANCE_MATRIX" **Parameter Options** - DISTANCE_MATRIX - RAW_DATA	The Hierarchical clustering method can be used on raw data as well as the data in Distance Matrix format. Pass the appropriate option to fit your dataset. If *Raw Data* is chosen, the similarity matrix will be computed before clustering is performed.

Algorithm Parameters: k-Means for Clustering

Parameters	Option Settings or Example	Explanation
maxIterations	"maxIterations" : 10	This parameter limits the number of iterations for the k-Means Clustering algorithm. Even if the convergence criteria has not yet been met, the cluster adjustment will stop once the limit on *# Iterations* has been reached. The default value for this option is 10.
numClusters	"numclusters" : 3	This parameter controls the number of final cohesive groups of observations (k) to be formed. The number of clusters should be at least 1 and at most the number of observations-1 in the data range. This value should be based on your knowledge of the data and the number of projected clusters. One can use the results of Hierarchical Clustering or several values of K to understand the best data partitioning level. The default value for this option is 2.
numStarts	"numStarts" : 5	If numStarts = 0, the clustering method will start building the model with a single fixed starting point. If numStarts is set to a positive integer, the algorithm will start at any random point. Enter the number of desired starting points for the clustering algorithm. The final result of the K-Means Clustering algorithm depends on the initial choice on the cluster centroids. The clustering algorithm allows a better choice by trying several random assignments. The best assignment (based on Sum of Squared Distances) is chosen as an initialization for further K-Means iterations.
randomSeed	"randomSeed" : 123	This option initializes the random number generator that is used to assign the initial cluster centroids. Setting the random number seed to a nonzero value (any number of your

		choice is OK) ensures that the same sequence of random numbers is used each time the initial cluster centroids are calculated. When the seed is not specified or is set to zero, the random number generator is initialized from the system clock, so the sequence of random numbers will be different each time the centroids are initialized.

Algorithm Parameters: Linear Regression

Parameters	Option Settings or Example	Explanation
`categoricalFeaturesNames`	`"categoricalFeaturesNames": [ 'X1' ]`	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.
`fitIntercept`	`"fitIntercept": false`	Set "fitIntercept":false to force the intercept term to 0. Set this option to "true", the default, to fit the Linear Regression intercept.
`weights`	`"weights": [1.0,2.1,...]`	Provides a weight variable allowing the user to allocate a weight to each record. A record with a large weight will influence the model more than a record with a smaller weight.

Algorithm Parameters: Latent Semantic Analysis for Text Mining

Parameters	Option Settings or Example	Explanation
`computeConceptImportance`	`"computeConceptImportance" : true`	If true, the Concept Importance table is computed. This table displays the total number of concepts extracted, the Singular Value for each, the Cumulative Singular Value and the % of Singular Value explained..
`computeTermImportance`	`"computeTermImportance" : true`	If true, the Term Importance table is computed. This table display each term along with its Importance as calculated by singular value decomposition. This option is not selected by default.
`maxNumConcepts`	`"maxNumConcepts": 4`	Use this parameter to specify the maximum number of concepts in the Scree Plot.
`minPercentExplained`	`"minPercentExplained" : 50`	Identifies the concepts with singular values that, when taken together, sum to the minimum percentage explained, 90% is the default.

Algorithm Parameters: TFIDF for Text Mining

Parameters	Option Settings or Example	Explanation

endPhrase	"endPhrase" : "End of Chat Transcript"	Text appearing before the first occurrence of the Start Phrase (if used) will be disregarded and similarly, text appearing after End Phrase (if used) will be disregarded. For example, if text mining the transcripts from a Live Chat service, you would not be particularly interested in any text appearing before the heading “Chat Transcript” or after the heading “End of Chat Transcript”. Thus you would pass “Chat Transcript” for "startPhrase" and “End of Chat Transcript” for "endPhrase".
emailToken	"emailToken" : "emailToken"	If NORMALIZE_EMAIL is passed for "processing", then this term may be used to replace email addresses appearing in the document collection with the term entered here. If "emailToken" is not specified, email addresses will be replaced with the phrase: "emailToken".
exclusiveInclusionTerms	"exclusiveInclusionTerms" : ["exclusiveInclusionTerm1", "exclusiveInclusionTerm2"]	If used, terms entered for "exclusiveInclusionTerms" will be removed from the document collection during pre-processing.
exclusionTerms	"exclusionTerms":["Term1", "Term2"]	If used, terms entered for "exclusionTerms" will be removed from the document collection during pre-processing.
phraseReplacement	" phraseReplacement": [["phrase1","phraseReplacement"], ["phrase2","phraseReplacement"]]	Use this parameter to combine words into phrases that indicate a singular meaning such as “station wagon” which refers to a specific type of car rather than two distinct tokens – station and wagon.
preprocessing	"preprocessing": ["REMOVE_STOPWORDS", "NORMALIZE_CASE", "STEM", "NORMALIZE_URL", "NORMALIZE_EMAIL", "NORMALIZE_NUMBER", "NORMALIZE_MONEY", "REMOVE_HTML_TAGS"],	Use this property to replace or remove nonsensical terms such as HTML tags, URLs, Email addresses, etc. from the document collection. It’s possible to remove normalized terms completely by including the normalized term (for example, “emailtoken”) in the Exclusion list. NORMALIZE_CASE – All text will be converted to a consistent (lower) case, so that Term, term, TERM, etc. are all normalized to a single token “term”. NORMALIZE_EMAIL -- Email addresses appearing in the document collection will be replaced with the term, “emailtoken”. NORMALIZE_MONEY -- Monetary amounts will be substituted with the term, “moneytoken”. NORMALIZE_NUMBER -- Numbers appearing in the document collection will be replaced with the term, “numbertoken”.

		NORMALIZE_URL -- URLs appearing in the document collection will be replaced with the term, “urltoken”. REMOVE_HTML_TAGS -- HTML tags will be removed from the document collection. REMOVE_STOPWORDS – Over 300 commonly used words/terms (such as a, to, the, and, etc.) will be removed from the document collection during preprocessing. Additional stopwords may be added or via a text document (*.txt). Terms in the text document can be separated by a space, a comma, or both. STEM -- If stemming reduced a term’s length to 2 or less characters, Text Miner will disregard the term.
`maxDocumentFrequency`	`"maxDocumentFrequency": 95`	Text Miner will remove terms that appear in more than the percentage of documents specified. The default percentage is 98%.
`maxVocabulary`	`"maxVocabulary": 5`	This parameter reduces the number of terms in the final vocabulary to the most frequently occurring in the collection. The default is “1000”.
`maxTermLength`	`"maxTermLength": 10`	Text Miner will remove terms that contain a set number of characters. This option can be extremely useful for removing some parts of text which are not actual English words, for example, URLs or computer-generated tokens, or to exclude very rare terms such as Latin species or disease names, i.e. Pneumonoultramicroscopicsilicovolcanoconiosis.
`moneyToken`	`"moneyToken" : "moneyToken"`	If NORMALIZE_MONEY is passed for "preprocessing", numbers appearing in the document collection will be replaced with the term, “numbertoken”.
`minDocumentFrequency`	`"minDocumentFrequency": 5`	Text Miner will remove terms that appear in less than the percentage of documents specified. The default percentage is 2%.
`minStemmedTermLength`	`"minStemmedTermLength": 2`	If stemming reduced a term’s length to 2 or less characters, Text Miner will disregard the term.
`numberToken`	`"numberToken":"numberToken"`	If NORMALIZE_NUMBER is passed for "processing", then this term may be used to replace numbers appearing in the document collection with the term entered here. If "numberToken" is not specified, numbers will be replaced with the phrase: "numberToken".

startPhrase	"startPhrase" : "Beginning of Chat Transcript"	Text appearing before the first occurrence of the Start Phrase (if used) will be disregarded and similarly, text appearing after End Phrase (if used) will be disregarded. For example, if text mining the transcripts from a Live Chat service, you would not be particularly interested in any text appearing before the heading “Chat Transcript” or after the heading “End of Chat Transcript”. Thus you would pass “Chat Transcript” for "startPhrase" and “End of Chat Transcript” for "endPhrase".
stopWordsExtraTerms	"stopWordsExtraTerms" : ["stopwords"]	Over 300 commonly used words/terms (such as a, to, the, and, etc.) will be removed from the document collection during preprocessing. Additional stop words may be added to the list using this parameter.
synonyms	"synonyms": [["rootTerm1", "synonym1", "synonym2"], ["rootTerm2", "synonym1", "synonym2"]],	Use this parameter to replace synonyms such as “car”, “automobile”, “convertible”, “vehicle”, “sedan”, “coupe”, “subcompact”, and “jeep” with “auto”. During pre-processing, Text Miner will replace the terms “car”, “automobile”, “convertible”, “vehicle”, “sedan”, “coupe”, “subcompact” and “jeep” with the term “auto”. It is possible to add synonyms from a text file.
urlToken	"urlToken":"urlToken"	If NORMALIZE_URL is passed for "preprocessing", URLS appearing in the document collection will be replaced with the term, “urlToken”.

Algorithm Parameters: ARIMA for Time Series

Parameters	Option Settings or Example	Explanation
autoRegressiveOrder	"autoRegressiveOrder" : 1	Sets the non-seasonal Autoregressive parameter (p).
Difference	"difference" : 1	Sets the non-seasonal Difference parameter (d).
Period	"period": 12	Enter the number of periods that make up one season.
maxIterations	"maxIterations" : 5	Sets the maximum number of iterations.
movingAverageOrder	"movingAverageOrder" : 2	Sets the non-seasonal Moving Average parameter (d).
seasonalAutoRegressiveOrder	"seasonalAutoRegresiveOrder": 1	Sets the seasonal Autoregressive parameter (P).
seasonalDifference	"seasonalDifference" : 1	Sets the seasonal Difference parameter (D).
seasonalMovingAverageOrder	"seasonalMovingAverageOrder": 1	Sets the seasonal Moving Average parameter (Q).

Algorithm Parameters: Lag Analysis for Time Series

Parameters	Option Settings or Example	Explanation
`maxLag`	`"maxLag" : 10`	Sets the maximum number of lags.
`minLag`	`"minLag" : 4`	Sets the minimum number of lags.
`nonSeasonalLag`	`"nonSeasonalLag": 1`	Sets the nonseasonal lag.
`period`	`"period" : 12`	Sets the number of periods that make up one season.
`seasonalLag`	`"seasonalLag" : 1`	Sets the seasonal lag.

Algorithm Parameters: Smoothing Methods for Time Series

Parameters	Option Settings or Example	Explanation
`interval`	`"interval" : 2` **Included in Following Methods** Moving Average	Use this parameter to enter the window width for the moving average smoothing method. This parameter accepts a value of 1 up to N -1 (where N is the number of observations in the dataset). If a value of 5 is entered for the Interval, then the average of the last five observations for the last smoothed point or $F_{t =}$ $(Y_t + Y_{t-1} + Y_{t-2} + Y_{t-3} + Y_{t-4})$ / 5 will be used. The default value is 2.
`levelParam`	`"levelParam" : 0.2` **Included in Following Methods** Double Exponential Exponential Holt Winters Additive Holt Winters Multiplicative Holt Winters No Trend	Use this parameter to enter the smoothing parameter for exponential, double exponential, and holt winters smoothing methods. This parameter is used in the weighted average calculation and can be from 0 to 1. A value of 1 or close to 1 will result in the most recent observations being assigned the largest weights and the earliest observations being assigned the smallest weights in the weighted average calculation. A value of 0 or close to 0 will result in the most recent observations being assigned the smallest weights and the earliest observations being assigned the largest weights in the weighted average calculation. The default is 0.2.
`Optimize`	`"optimize": true` **Included in Following Methods** Double Exponential Exponential Holt Winters Additive Holt Winters Multiplicative Holt Winters No Trend	Select this option to select the Alpha parameter for the Exponential smoothing method, the Alpha and Beta parameters for the Double Exponential Smoothing method, and the Alpha, Beta, and Gamma parameters for the Holt Winter Smoothing method to minimize the residual mean squared errors in the training and validation sets. Take care when using this feature as this option can result in an over fit model. This option is not turned on by default.
`Period`	`"period" : 12` **Included in Following Methods**	Enter the number of periods that make up one season when using the Holt Winter Smoothing method.

	Holt Winters Additive Holt Winters Multiplicative Holt Winters No Trend	
seasonalityParam	"seasonalityParam" : 0.05 **Included in Following Methods** Holt Winters Additive Holt Winters Multiplicative Holt Winters No Trend	The Holt Winters Smoothing technique utilizes an additional seasonal parameter, Gamma, to manage the presence of seasonality in the data. This parameter is also used in the weighted average calculation and can be from 0 to 1. A value of 1 or close to 1 will result in the most recent observations being assigned the largest weights and the earliest observations being assigned the smallest weights in the weighted average calculation. A value of 0 or close to 0 will result in the most recent observations being assigned the smallest weights and the earliest observations being assigned the largest weights in the weighted average calculation. The default is 0.05.
trendParam	"trendParam" : 0.15 **Included in Following Methods** Double Exponential Holt Winters Additive Holt Winters Multiplicative	The Double Exponential and Holt Winters smoothing techniques include the parameter, Beta, to contend with trends in the data. This parameter is also used in the weighted average calculation and can be from 0 to 1. A value of 1 or close to 1 will result in the most recent observations being assigned the largest weights and the earliest observations being assigned the smallest weights in the weighted average calculation. A value of 0 or close to 0 will result in the most recent observations being assigned the smallest weights and the earliest observations being assigned the largest weights in the weighted average calculation. The default is 0.15.

Algorithm Parameters: Binning for Transformation

Parameters	Option Settings or Example	Explanation
binValueOption	"binValueOption": [["x4", "RANK"], ["x2", "MID_VALUE"]] **Parameter Options** • MEAN • MEDIAN • MID_VALUE • RANK	When method = EQUAL_COUNT, use MEAN to replace the value of the selected variable with the mean of the interval for the assigned bin. When method = EQUAL_COUNT, use MEDIAN to replace the value of the selected variable value with the median of the interval for the assigned bin. When method = EQUAL_INTERVAL, use MID_VALUE to replace the value of the selected variable with the mid value of the interval for the assigned bin. When method = EQUAL_INTERVAL *or EQUAL_COUNT*, use RANK to specify the

		Start value of the first bin and the *Interval* of each bin. Subsequent bin values will be calculated as the previous bin + interval value.
interval	"interval": [["x2", "RIGHT_CLOSED"], ["x4", "CLOSED"]]	Use this parameter to indicate whether the interval for each variable is CLOSED [], RIGHT_CLOSED (], or LEFT_CLOSED [). Default is: LEFT_CLOSED [).
method	"method": [["x2", "EQUAL_INTERVAL"], ["x4", "EQUAL_COUNT"]]	EQUAL_COUNT -- Data is binned in such a way that each bin contains the same number of records. The options for the value of the binned variable for this parameter are Rank, Mean, and Median. Note: There is a possibility that the number of records in a bin may not be equal due to factors such as border values, the number of records being divisible by the number of bins, etc. EQUAL_INTERVAL -- Binning procedure will assign records to bins if the record's value falls in the interval of the bin. Bin intervals are calculated by roughly subtracting the Minimum variable value from the Maximum variable value and dividing by the number of bins ((Max Value – Min Value) / # bins). The options for value of the binned variable for this process are *Rank* and *Mid value*.
numBins	"numBins": [["x2", 11], ["x4", 4]]	Enter the number of desired bins for each selected variable using this option.
rank	"rank": [["x4", 1.0, 5.0]]	This parameter is available when binValueOption is set to "RANK". Use the "rank" parameter to specify the *Start* value of the first bin and the *Interval* of each bin. Subsequent bin values will be calculated as the previous bin + interval value.

Algorithm Parameters: Canonical Variate Analysis for Transformation

No Parameters are associated with this Transformation Method

Algorithm Parameters: Factorization for Transformation

Parameters	Option Settings or Example	Explanation
baseIndex	"baseIndex": [["X1", 1], ["X3", 5]]	Factorization converts a variable into a new numeric, categorical variable. Use this parameter to specify the number with which to begin the categorization. In the example to the left, the categorization will begin at "1" for "X1" (1, 2, 3, 4, 5,) and "5" for "X3" (5, 6, 7, 8, 9).

Algorithm Parameters: Imputation for Transformation

Parameters	Option Settings or Example	Explanation
imputationStrategy	"imputation": [["A", "MODE"], ["B", "MEAN"]] **Parameter Options** • DELETE_RECORD • MEAN • MEDIAN • MODE • VALUE	Determines the strategy of the selected variables. Delete record **-** Deletes the entire record if a missing or invalid value is found for that variable. Mean **-** All missing values in the column for the variable specified will be replaced by the mean **-** the average of the values in the remainder of the column. Median **-** All missing values in the column for the variable specified will be replaced by the median **-** the number that would appear in the middle of the remaining column values if all values were written in ascending order. Mode **-** All missing values in the column for the variable specified will be replaced by the mode **-** the value occurring most frequently in the remainder of the column. Value – Use VALUE to enter a user defined value. If used, the imputation model parameter must be used to specify the user defined value.
placeholder	"placeholder" : [["B", 1000.0], ["F", "unknown"], ["G", -1]]	Sets the custom placeholder for missing values in a column. Note that integer columns may not contain \`NaN\`, \`null\` or any other invalid value by default, but can still specify a manual placeholder for missing integer values. See Imputation Model Parameters below for an example or open the Imputation Example from the Editor tab at RASON Examples – Data Mining – Transformation – Imputation.

Algorithm Parameters: Linear/Logistic Wrapping for Transformation

Parameters	Option Settings or Example	Explanation
fIn	"fIn" : 3.84	Used when method = FORWARD_SELECTION or STEPWISE_SELECTION A statistic is calculated when variables are added or eliminated. For a variable to come into the regression, the statistic's value must be greater than the value for fIn (default = 3.84).
fitIntercept	"fitIntercept" : true	Fits the Linear/Logistic Regression intercept. If this option is set to False, the intercept term is forced to 0.

`fOut`	`"fOut" : 2.71`	For use when method = BACKWARD_ELIMINATION OR STEPWISE_SELECTION. A statistic is calculated when variables are eliminated. For a variable to leave the regression, the statistic's value must be less than the value of FOUT (default = 2.71).
`maxNumSubsetsExhaustive`	`"maxNumSubsetsExhaustive" : 3`	For use when method = EXHAUSTIVE_SEARCH. Enter an integer value for the maximum number of subsets.
`maxSubsetSize`	`"maxSubsetSize" : 4`	Enter an integer from 1 up to N where N is the number of variables (features) in the model.
`method`	`"method" : "BACKWARD_ELIMINATION"` **Parameter Options** • `BACKWARD_ELIMINATION` • `EXHAUSTIVE_SEARCH` • `FORWARD_SELECTION` • `SEQUENTIAL_REPLACEMENT` • `STEPWISE_SELECTION`	Five different selection procedures are available for selecting the best subset of variables. *Backward Elimination* in which variables are eliminated one at a time, starting with the least significant. If this procedure is selected, use the "fOut" parameter to set the statistic to determine when a variable is to be eliminated. *Forward Selection* in which variables are added one at a time, starting with the most significant. If this procedure is selected, use the "fIn" parameter to set the statistic to determine when a variable is to come into the regression. *Sequential Replacement* in which variables are sequentially replaced and replacements that improve performance are weights retained. *Stepwise selection* is similar to Forward selection except that at each stage, variables that are not statistically significant may be dropped. When this procedure is selected, the Stepwise selection options "fIn" and "fOut" are enabled. *Exhaustive Search* where searches of all combinations of variables are performed to observe which combination has the best fit. (This option can become quite time consuming depending on the number of input variables.) If this procedure is selected, use "maxNumSubsetsExhaustive" to set the maximum number of best subsets.
`numTopFeatures`	`"numTopFeatures" : 2`	Model option only. Enter a value ranging from 1 to the number of features in the model.

Algorithm Parameters: One Hot Encoding for Transformation

Parameters	Option Settings or Example	Explanation
`categoricalFeaturesNames`	`"categoricalFeaturesNames": [ "X1" ]`	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.

Algorithm Parameters: PCA for Transformation

Parameters	Option Settings or Example	Explanation
`matrixMethod`	`"matrixMethod": "CORRELATION"` **Parameter Options** • CORRELATION • COVARIANCE	When computing Principal Components, the data is matrix multiplied by a transformation matrix. Use this option to specify the method used to calculate this transformation matrix. **CORRELATION** An alternative method is to derive the transformation matrix on the eigenvectors of the correlation matrix instead of the covariance matrix. *The correlation matrix is equivalent to a covariance matrix for the data where each variable has been standardized to zero mean and unit variance.* This method tends to equalize the influence of each variable, inflating the influence of variables with relatively small variance and reducing the influence of variables with high variance. This option is selected by default. **COVARIANCE** The covariance matrix is a square, symmetric matrix of size n x n (number of variables by number of variables). The diagonal elements are variances and the off-diagonals are covariances. The eigenvalues and eigenvectors of the covariance matrix are computed and the transformation matrix is defined as the transpose of this eigenvector matrix. *If the covariance method is selected, the dataset should first be normalized*. One way to organize the data is to divide each variable by its standard deviation. Normalizing gives all variables equal importance in terms of variability.[6]
`numPrincipalComponents`	`"numPrincipalComponents": 2`	This option is mutually exclusive with varianceCutoff. Use either numPrincipalComponents to select the number of principal components displayed in the output or varianceCutoff, but not both.

[6] Shmueli, Galit, Nitin R. Patel, and Peter C. Bruce. Data Mining for Business Intelligence. 2nd ed. New Jersey: Wiley, 2010

		This option specifies a fixed number of components. Enter an integer value from 1 to n where n is the number of *Input variables* in the model.
`varianceCutoff`	`"varianceCutoff" : 0.98`	This option is mutually exclusive with numPrincipalComponents. Use either numPrincipalComponents to select the number of principal components displayed in the output or varianceCutoff, but not both. Use this option to calculate the minimum number of principal components required to account for the percentage of variance entered for this option.

Algorithm Parameters: Rescaler for Transformation

Parameters	Option Settings or Example	Explanation
`correction`	`"correction": 0.01`	Sets the "correction" option when technique = "NORMALIZATION" or "ADJUSTED_NORMALIZATION". See "technique" explanation below.
`excludedCols`	`"excludedCols": ["CHAS", "MEDV"]`	Excludes specified columns from the data mining method. (Array property)
`normType`	`"normType":"L1"`	If technique = "UNIT_NORMALIZATION", use "normType" to set the normalization type. "L1" normalizes the data using the Manhattan Distance (L1-norm) while "L2" uses the Euclidean Length (L2-norm).
`technique`	`"technique": "STANDARDIZATION"`	The following methods for feature scaling are: • STANDARDIZATION makes the feature values have zero mean and unit variance. (x−mean)/std.dev. • NORMALIZATION scales the data values to the [0,1] range. (x−min)/(max−min) The Correction option specifies a small positive number ε that is applied as a correction to the formula. The corrected formula is widely used in Neural Networks when Logistic Sigmoid function is used to activate the neurons in hidden layers – it ensures that the data values never reach the asymptotic limits of the activation function. The corrected formula is [x−(min−ε)]/[(max+ε)−(min−ε)]. To set the Correction option use: "correction". • ADJUSTED_NORMALIZATION scales the data values to the [-1,1] range. [2(x−min)/(max−min)]−1 The Correction option specifies a small positive number ε that is applied as a correction to the formula. The corrected formula is widely used in

		Neural Networks when Hyperbolic Tangent function is used to activate the neurons in hidden layers – it ensures that the data values never reach the asymptotic limits of the activation function. The corrected formula is $\{2[(x-(min-\varepsilon))/((max+\varepsilon)-(min-\varepsilon))]\}-1$. To set the Correction option use: "correction". •UNIT_NORMALIZATION is another frequently used method to scale the data such that the feature vector has a unit length. This usually means dividing each value by the Euclidean length (L2-norm) of the vector. In some applications, it can be more practical to use the Manhattan Distance (L1-norm).

Algorithm Parameters: Univariate for Transformation

Parameters	Option Settings or Example	Explanation
`binningTypeFeatures`	`"binningTypeFeatures": "EQUAL_INTERVAL"` **Parameter Options** • `EQUAL_COUNT` • `EQUAL_INTERVAL` • `NONE`	This parameter transforms continuous input variables into categorical variables. Records are assigned to the bins based on the variable's value. Use EQUAL_INTERVAL, if the value falls within the interval of the bin, or EQUAL_COUNT, if there is an equal number of records in each bin. These settings will be applied to each continuous variable in the model. If NONE is selected, the variable will not be discretized.
`binningTypeTarget`	`"binningTypeTarget": "EQUAL_COUNT"` **Parameter Options** • `EQUAL_COUNT` • `EQUAL_INTERVAL` • `NONE`	This parameter transform a continuous output variable into a categorical variable. Records are assigned to the bins based on the variable's value. Use EQUAL_INTERVAL, if the value falls within the interval of the bin, or EQUAL_COUNT, if there is an equal number of records in each bin. If NONE is selected, the variable will not be discretized.
`categoricalFeaturesNames`	`categoricalFeaturesNames: [ 'X1' ]`	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.
`categoricalTarget`	`"categoricalTarget" : true`	True – Setting this option to "true" denotes that the Output Variable is a categorical variable. False – Setting this option to "false" denotes that the Output Variable is a continuous variable.
`metric`	`"metric" : "CHI2"` **Parameter Options** • `CHI2`	Use this parameter to compute the desired metric for Feature Selection. CHI2 -- Used to assess the statistical independence of two events. When applied to

	- CRAMERSV - FISHER - FTEST - GAINRATIO - GINI - KENDALL - MUTUALINFO - PEARSON - SPEARMAN - WELCH	Feature Selection, it is used as a test of independence to assess whether the assigned class is independent of a particular variable. The minimum value for this statistic is 0. The higher the Chi-Squared statistic, the more independent the variable. CRAMERSV -- Variation of the Chi-Squared statistic that also measures the association between two discrete nominal variables. This statistic ranges from 0 to 1 with 0 indicating no association between the two variables and 1 indicating complete association (the two variables are equal). FISHER -- Variation of the F-Statistic. It choose assigns higher values) to variables that assign si values to samples from the same class and differ values to samples from different classes. The lar Fisher Score value, the more relevant or importa variable (or feature). FTEST -- Tests the hypothesis of at least one sa mean being different from other sample means assuming equal variances among all samples. If variance between the two samples is large with to the variance within the sample, the F statistic large. Specifically for Feature Selection purpose used to test if a particular feature is able to separ records from different target classes by examinin between-class and within-class variances. GAINRATIO -- Ranging from 0 and 1, is defined as the mutual information (or information gain) normalized by the feature entropy. This normalization helps address the problem of overemphasizing features with many values but the normalization results in an overestimate of the relevance of features with low entropy. It is a good practice to consider both mutual information and gain ratio for deciding on feature rankings. The larger the gain ratio, the larger the evidence for the feature to be relevant in a classification model. GINI -- Measures a variable's ability to distingu between classes. The maximum value of the ind binary classification is 0.5. The smaller the Gin the more relevant the variable. KENDALL -- Also known as Kendall's tau coef is also used to measure the level of association between two variables. A tau value of +1 signif perfect agreement and a -1 indicates complete disagreement. If a variable and the outcome var are independent, then one could expect the Kend to be approximately zero.

		MUTUALINFO -- The degree of a variables' mutual dependence or the amount of uncertainty in variable 1 that can be reduced by incorporating knowledge about variable 2. Mutual Information is non-negative and is equal to zero if the two variables are statistically independent. Mutual Info is always less than the entropy (amount of information contained) in each individual variable. PEARSON -- A widely used statistic that measures the closeness of the linear relationship between two variables, with a value between +1 and −1 inclusive, where 1 indicates complete positive correlation, 0 indicates no correlation, and −1 indicates complete negative correlation. SPEARMAN -- A nonparametric measure that assesses the relationship between two variables. This measure calculates the correlation coefficient between the ranked values of the two variables. If data values are repeated, the Spearman rank correlation coefficient will be +1 or -1, if each of the variables is a perfect monotone (or non-varying) function of the other. WELCH -- A two-sample test (i.e. applicable for binary classification problems) that is used to check the hypothesis that two populations with possibly unequal variances have equal means. When used with the Feature Selection tool, a large T-statistic value (in conjunction with a small p-value) would provide sufficient evidence that the Distribution of values for each of the two classes are distinct and the variable may have enough discriminative power to be included in the classification model.
`numBinsFeatures`	`"numBinsFeatures": 5`	Sets the maximum number of bins for the input variables.
`numBinsTarget`	`"numBinsTarget": 3`	Sets the maximum number of bins for the target (output) variable.

Transform ("transformer")

The "transformer" object is used to differentiate the algorithms that do not have a model, i.e. they do not implement the "fit" interface or extract any type of model. Rather, these algorithms implement the "transform" interface (only) by operating directly on the data to produce transformed data which can serve as input to other data mining methods. Since no data is stored (i.e. data in, data out), transformation algorithms are represented by a single object, for example: dataFrame df = Sampler::transform(data).

Data mining algorithms that do not "fit" a model are: Partitioning, Sampling, Big Data (Sampling and Summarizing), Association Rules and Feature Selection.

```
transformer: {
    mySampler: {
      type: 'transformation',
      algorithm: 'sampling',
      parameters: {
        sampleSize: 4,
        replaceOption: false,
        sortIndexes: false,
        seed: 123
      }
    }
  },
```

The following properties are available for use in this section.

Property	Example	Definition
`algorithm`	`"algorithm": "associationRules"`	Available option settings will vary depending on the type setting. Use this property to specify the algorithm to be used to perform the transformation. See the chart below for all options.

`parameters`	`"parameters" : {`
	`method: 'M1_BREIMAN',`
	`numWeakLearners: 2,`
	`resamplingSeed: 10`
	`}`
		Available parameter settings will vary depending on the algorithm setting. Use this property to set parameter values or turn parameters on or off using "true" or "false". For a full list of parameters, see below.

`type`	`"type":"transformation"` **Parameter Options** • `affinityAnalysis` • `bigData` • `featureSelection` • `transformation`	Use this property to specify the type of transformative algorithm to be applied: affinity analysis, big data, feature selection, or transformation.

The chart below contains the available options for "algorithm" based on the "type" argument.

If "type" =	Algorithm option settings	Definition
`affinityAnalysis`	`"algorithm" : "associationRules"`	Runs Association Rules method.
`bigData`	`"algorithm" : "sampling"` **Parameter Options**	"sampling" – Use to sample from Big Data. "summarization" – Use to summarize from Big Data.

	• sampling • summarization	
featureSelection	"algorithm": "univariate " **Parameter Options** • linearWrapping • logisticWrapping • univariate	"univariate" – Performs Feature Analysis by ranking variables according to one or more univariate measures. "linearWrapping" – Performs Feature Selection (on a continuous output variable) using Linear Wrapping. "logisticWrapping" - Performs Feature Selection (on a categorical output variable) using Logistic Wrapping.
transformation	"algorithm" : "sampling" **Parameter Options** • categoryReduction • oversamplePartitioning • partitioning • sampling • stratifiedSampling	"categoryReduction" – Converts a string variable into a new numeric, categorical variable. "oversamplePartitioning"- Partitioning allowing oversampling. "partitioning" – Partitioning where every observation in the main dataset has equal probability of being selected for the partition dataset. "sampling" – Draws a representative sample from a dataset. "stratifiedSampling" - The population is first divided into groups of similar items, called strata. Each stratum, in turn, is sampled using simple random sampling. These samples are then combined to form a stratified random sample.

The chart below contains the available options for "parameters" based on the "algorithm" argument.

Algorithm Parameters: Association Rules for Affinity Analysis

Property	Example	Definition
method	"method": "T_TREE" **Parameter Options** • APRIORI • T_TREE	APRIORI – Use this option to use the APRIORI algorithm to find all frequent item sets in a database. T_TREE – Use this option to use the T-Tree algorithm to find all frequent item sets in a database.
minSupport	"minSupport": 0.1	Specify the minimum number of transactions in which a particular item-set must appear for this set to qualify for inclusion in an association rule here. The default value is 10% of the total number of rows.

Algorithm Parameters: Common Sampling Options for Big Data

Property	Example	Definition

async	"async": true	Submits Big Data job asynchronously. Get Job ID for later retrieval.
awsS3	"awsS3": true	If data source is Amazon S3 (AWS S3), set this option to true.
awsS3AccessKey	"awsS3AccessKey": "<AWS S3 access key>"	Passes the Amazon S3 access key.
awsS3SecretKey	"awsS3SecretKey": "<AWS S3 secret key>"	Passes the Amazon S3 secret key.
dataFormat	"dataFormat":"PARQUET"	If data is in Apache Parquet format, use "PARQUET" for this option. If your data is in Delimited Text format, USE "CSV". If "dataformat: "CSV", use "**headerExists**":true (the default) to specify that the first row in your dataset contains headers. Use "delimiter" property to specify the delimiter used in the CSV file.
delimiter	"delimiter": ";" **Parameter Options** • Comma - "," • Other - "<other>" • Semicolon - ";" • Space - "" • Tab – "\t"	Use this option to specify the delimiter used in the CSV file.
fileLocation	"fileLocation": "<file location URL - hdfs://..., s3n://...>"	Enter the location of the Big Data file here.
headerExists header	"headerExists": true "header": true	Set to True by default. This option indicates that the first row in the CSV file includes file headings.
jobID	"jobID": '<job ID of previously submitted async job>'	Enter the ID of the previously submitted async job.
selectedVariables	"selectedVariables": ['Var1', 'Var2', 'Var3']	Variables passed to this parameter will be included in the sample.
sparkServer	"sparkServer": "<endpoint for Spark cluster>"	Use this option to enter the endpoint for the Apache Spark cluster. Note: port for the Spark REST server must be 8090'.

Algorithm Parameters: Sampling for Big Data

Property	Example	Definition
randomSeed	"randomSeed": 123	Sets the desired sorting seed here. Setting the random number seed to a nonzero value ensures that the same sequence of random numbers is used each time the dataset is chosen for sampling. The default seed is 12345.

sampleFraction	"sampleFraction":0.01	This is the expected size of the sample as a fraction of the dataset's size. If "withReplacement": true, the value for "sampleFraction" must be greater than 0. If "withReplacement": false, the setting for "sampleFraction" becomes the probability that each element is chosen. As a result, "*sampleFraction"* must be between 0 and 1.
sampleSize	"sampleSize": 100	Sets the desired sample size here. (Note that the actual sample size in the output may vary a little, depending on additional options selected.)
samplingType	"samplingType": APPROXIMATE **Parameter Options** • "APPROXIMATE" • "EXACT"	"APPROXIMATE" -- When this option is selected, the size of the resultant sample will be determined by the value entered for "sampleFraction". "EXACT" -- When this option is selected, a fixed – size sampled subset of data, determined by "sampleSize", is returned.
trackRowID	"trackRowID":true	If this option is set to "true", data records in the resulting sample will carry the ordinal IDs corresponding to the original data records. Note: Selecting this option may significantly increase running time.
withReplacement	"withReplacement": true	If this option is set to "true" the data will be sampled with replacement. The default is sampling without replacement ("withReplacement": false).

Algorithm Parameters: Summarization

Property	Example	Definition
aggregationType	"aggregationType": "SUM" **Parameter Options** • "AVG" • "MAX" • "MIN" • "STDDEV" • "SUM"	This option provides 5 statistics that can be inferred from the dataset: sum, average, standard deviation, minimum and maximum.
computeGroupCounts	"computeGroupCounts" : true	Use this option when 1 or more Grouping Variables exist. When this option is set to

		"true", the number of records belonging to each group is computed and reported.
`groupingVariables`	`"groupingVariables": ['Var1']`	Use this option to specify a grouping variable(s).

Algorithm Parameters: Linear/Logistic Wrapping for Transformation

Parameters	Option Settings or Example	Explanation
`categoricalFeaturesNames`	`categoricalFeaturesNames: [ 'X1' ]`	Enter categorical variables by name using this parameter. Any non-numeric columns are automatically considered as categorical (nominal) variables.
`fIn`	`"fIn" : 3.84`	Used when method = FORWARD_SELECTION or STEPWISE_SELECTION A statistic is calculated when variables are added or eliminated. For a variable to come into the regression, the statistic's value must be greater than the value for FIN (default = 3.84).
`fitIntercept`	`"fitIntercept" : true`	Fits the Linear/Logistic Regression intercept. If this option is set to False, the intercept term is forced to 0.
`fOut`	`"fOut" : 2.71`	For use when method = BACKWARD_ELIMINATION OR STEPWISE_SELECTION. A statistic is calculated when variables are eliminated. For a variable to leave the regression, the statistic's value must be less than the value of FOUT (default = 2.71).
`maxNumSubsetsExhaustive`	`"maxNumSubsetsExhaustive" : 3`	For use when method = EXHAUSTIVE_SEARCH. Enter an integer value for the maximum number of subsets.
`maxSubsetSize`	`"maxSubsetSize" : 4`	Enter an integer from 1 up to N where N is the number of variables (features) in the model.
`method`	`"method" : "BACKWARD_ELIMINATION"` **Parameter Options** • BACKWARD_ELIMINATION • EXHAUSTIVE_SEARCH • FORWARD_SELECTION • SEQUENTIAL_REPLACEMENT • STEPWISE_SELECTION	Five different selection procedures are available for selecting the best subset of variables. *Backward Elimination* in which variables are eliminated one at a time, starting with the least significant. If this procedure is selected, use the FOUT parameter to set this statistic. *Forward Selection* in which variables are added one at a time, starting with the most significant. If this procedure is selected, use the fIn parameter to set this statistic.

		Sequential Replacement in which variables are sequentially replaced and replacements that improve performance are weights retained. *Stepwise selection* is similar to Forward selection except that at each stage, variables that are not statistically significant may be dropped. Use the fIn and fOut parameters to set these statistics. *Exhaustive Search* where searches of all combinations of variables are performed to observe which combination has the best fit. (This option can become quite time consuming depending on the number of input variables.) If this procedure is selected, use maxSubsetSize to set the maximum subset size.
`maxIterations`	`"maxIterations":5`	**For Logistic Wrapping Only** Sets the maximum number of iterations.
`weights`	`"weights": [1.0,2.1,...]`	Provides a weight variable allowing the user to allocate a weight to each record. A record with a large weight will influence the model more than a record with a smaller weight.

Algorithm Parameters: Sampling/Stratified Sampling for Transformation

Property	Example	Definition
`replaceOption`	`"replaceOption":false`	Set this option to "true" to sample with replacement. The default is sampling without replacement.
`sampleSize`	`"sampleSize":100`	Enter the desired sample size here. (Note that the actual sample size in the output may vary from the number entered here, depending on additional options selected.)
`Seed`	`"seed":123`	Enter the desired sorting seed here. The default seed is 12345.
`sortIndexes`	`"sortIndexes":true`	When this option is set to "true", the data is sorted using the simple random sampling technique, taking into account the additional parameter settings.

`stratificationMethod`	`"stratificationMethod":"PROPORTIONAL"`
	`Parameter Options`
	• `"EQUAL_SIZE"`
	• `"PROPORTIONAL"`
		PROPORTIONAL -- Detects the proportion of each stratum in the dataset and maintains the same in sampling. At time, the sample size must be increased in order to maintain the proportionate stratum size.
		EQUAL_SIZE -- Generates a sample using the same number of records from each stratum. The number passed for

		"stratumSampleSize" automatically decides the desired sample size.
`stratumSampleSize`	`"stratumSampleSize":10`	Sets desired stratum sample size.

Algorithm Parameters: Partitioning for Transformation

Property	Example	Definition
`partitionMethod`	`"partitionMethod":"RANDOM"` **Parameter Options** • MANUAL • RANDOM • SEQUENTIAL	MANUAL – Use this option when partitioning the dataset using a partition variable. (See partitionVariables below.) RANDOM – Use "partitionMethod":"RANDOM" to perform standard random sampling, where random observations are selected to be included in the training, validation, and test sets. SEQUENTIAL – use "partitionMethod": "SEQUENTIAL" to perform sequential partitioning.
`partitionVariable`	`partitionVariable: ["t", "t", "t", "v", "v", "v", "s", "s"]`	When "partitionMethod" = "Manual", the partition variable specified is used to partition the dataset which serves as a flag for writing each observation to the appropriate partition(s). This is useful when you have already predetermined the observations to be used in the training, validation, and/or test sets. This partition variable takes the value: "t" for training, "v" for validation and "s" for test. Rows with any other values in the Partition Variable column are ignored.

`ratios`	`"ratios": [`
	`   [ "Training", 0.5 ],`
	`   [ "Validation", 0.3 ],`
	`   [ "Testing", 0.2 ]`
	`]`
		Specify the percentages for the training set, validation set and test sets.

`seed`	`"seed":123`	Random partitioning uses the system clock as a default to initialize the random number seed. By default, this option is selected to specify a seed for random number generation for the partitioning. Setting this option will result in the same records being assigned to the same set on successive runs. The default seed entry is 12345.

Algorithm Parameters: Oversample Partitioning for Transformation

Property	Example	Definition
`seed`	`"seed":123`	Random partitioning uses the system clock as a default to initialize the random number seed. This option is not selected by default. Setting this option will result in the same records

		being assigned to the same set on successive runs. The default seed entry is 12345.
`successClass`	`"successClass":"1"`	Select the success value for the output variable here (i.e. 0 or 1 or “yes” or “no”).
`successRatioInTraining`	`"successRatioInTraining":0.5`	Sets the percentage of successes to be assigned to the training set. The default is 50%. With the default setting, 50% of the successes will be assigned to the training set and 50% will be assigned to the validation set.
`testRatioFromValidation`	`"testRatioFromValidation":0.1`	If a test set is desired, specify the percentage of the validation set that should be allocated to the test set using this option.

Algorithm Parameters: SyntheticDataGenerator for Transformation

Property	**Example**	**Definition**
Metalog Distribution Fitting Options		
`computeMetalogCurves`	`“computeMetalogCurves”: true`	Set computeMetalogCurves to true to compute Metalog PDF curves the selected Metalog distribution for all columns.

`metalogAuto`	`“metalogAuto”: true`
	`“metalogAuto”: false`
		If **False**, RASON Decision Services attempts to fit the Metalog distribution with the specified number of terms.
		If **True**, Rason Decision Services will attempt to fit *all possible* Metalog distributions, with the number of terms limited by the specified value, and select the best Metalog distribution according to the chosen Goodness-of-Fit test.

`MetalogGoodnessOfFitType`	`"metalogGoodnessOfFitType": [`
	`["CRIM", "CHI_SQUARE"],`
	`["ZN","KOLMOGOROV_SMIRNOFF"],`
	`["INDUS", "ANDERSON_DARLING"],`
	`["NOX", "AIC"],`
	`["RM", "AICc"],`
	`["DIS", "BIC"],`
	`["AGE", "BICc"],`
	`["TAX", "MAX_LIKELIHOOD"],`
	`],`
	
	`Type = CHI_SQUARE, KOLMOGOROV_SMIRNOV, ANDERSON_DARLING, AIC, BIC, AICc, BICc, MAX_LIKELIHOOD`
		The Goodness of Fit test is used to select the best Metalog form for each column among the candidate distributions defined by a different number of terms, from 2 to the value passed for NumMetalogTerms. The default Goodness-of-Fit test is Anderson-Darling.
		XLMiner SDK offers the following Goodness of Fit Tests:
		• Chi Square – Uses the chi-square statistic to rank the distributions. Sample data is first divided into intervals using either equal probability, then the number of points that fall into each interval are compared with the expected number of

		points in each interval. The null hypothesis is rejected using a 90% significance level, if the chi-squared test statistic is greater than the critical value statistic. • Kolmogorov-Smirnoff –This test computes the difference (D) between the continuous distribution function (CDF) and the empirical cumulative distribution function (ECDF). The null hypothesis is rejected if, at the 90% significance level, D is larger than the critical value statistic. • [Default] Anderson -Darling –Ranks the fitted distributions using the Anderson Darling statistic, A2 . The null hypothesis is rejected using a 90% significance level, if A2 is larger than the critical value statistic. This test awards more weight to the distribution tails then the Kolmogorov-Smirnoff test. • AIC – The AIC test is a Chi Squared test corrected for the number of distribution parameters and sample size. AIC = Chi-Square Statistic + 2 * k + 2 * k * (k + 1) / (n – k – 1) where k is the number of distribution parameters and n is the sample size. • AICc –When the sample size is small, there is a significant chance that the AIC test will select a model with many parameters. In other words, AIC will overfit the data. AICc was developed to reduce the possibility of overfitting by applying a penalty to the number of parameters. Assuming that the model is univariate, is linear in the parameters and has normally distributed residuals, the formula for AICc is: AICc = AIC + 2k 2+2k n−k−1 where n = sample size, k = # of parameters. As the sample size approaches infinity, the penalty on the number of parameters converges to 0 resulting in AICc converging to AIC. • BIC – The Bayesian information criterion (BIC) is defined as: BIC = k ln(n) = 2 ln ($\hat{L}$) where $\hat{L}$ = the maximized value of the likelikhood function of the model M. $\hat{L}$ = $p(x\|\theta, M)$ where θ are the parameter values that maximize the likelihood function and x is the observed data. n = Sample size k = Number of parameters

		BICc – The BICc is the alternative version of BIC, corrected for the sample size BICc = BIC + 2 * p * (p + 1) / (n – p - 1) • Maximum Likelihood (ML) – The (negated) raw value of the estimated maximum log likelihood utilized in tests described above.

metalogLowerBound[colname, lowerBound]	"metalogLowerBound": [
	     ["CRIM", 0.00632],
	     ["ZN", 0],
	     ["INDUS", 0.46],
	     ["NOX", 0.385],
	     ["RM", 3.561],
	     ["DIS", 1.1296],
	     ["AGE", 2.9],
	     ["TAX", 187],
	     ["PTRATIO", 12.6],
	     ["B", 0.32],
	     ["LSTAT", 1.73],
	     ["MEDV", 5]
	],
		Manually sets the lower bound for the Metalog distribution

metalogUpperBound[colname, lowerBound]	"metalogUpperBound": [
	     ["ZN", 100],
	     ["INDUS", 27.74],
	     ["NOX", 0.871],
	     ["RM", 8.78],
	     ["DIS", 12.1265],
	     ["AGE", 100],
	     ["TAX", 711],
	     ["PTRATIO", 22],
	     ["B", 396.9],
	     ["LSTAT", 37.97],
	     ["MEDV", 50]
	],
		Manually sets the upper bound for the Metalog distribution

numMetalogTerms	"numMetalogTerms": [
	    ["CRIM", 5],
	    ["ZN", 5],
	    ["INDUS", 5],
	    ["NOX", 5],
	    ["RM", 5],
	    ["DIS", 5],
	    ["AGE", 5],
	    ["TAX", 5],
	    ["PTRATIO", 5],
	    ["B", 5],
	    ["LSTAT", 5],
	    ["MEDV", 5]
		If “metalogAuto”: false, sets the number of terms for the Metalog Distributions
		If metalogAuto”: true, sets the max number of terms for the Metalog distribution for a given column.

	`],`	

`UseMinMaxAsBounds`	`UseMinMaxAsBounds = True`
	`UseMinMaxAsBounds = False`
		`True` sets the lower/upper bounds as minimum/maximum of each variable. However, if a lower or upper bound is manually set, RASON Decision Services will give priority to the manually set bounds while keeping the minimum/maximum for those variables where the bounds were not set manually. In other words, if a bound has been set manually, UseMinMaxAsBounds will not overwrite the existing bound.

Correlation Fitting Options		

CorrelationType	`“CorrelationType”: “None”`
	`“CorrelationType”: “RANK”`
	`“CorrelationType”: “COPULA”`
	Example: The priority of the copulas given in the example code below is 1. Clayton, 2. Frank, 3. Gumbel, 4. Gauss and 5. Student (order as listed).
	`“claytonCopula”: true`
	`“frankCopula”: true`
	`“gumbelCopula”: true`
	`“gaussCopula”: true`
	`“studentCopula”: true`
		Use `CorrelationType` to fit a correlation between the variables. If this option is set to “None”, then no correlation fitting will be performed.
		Otherwise, there are two options for correlation fitting: rank and copula.
		If *Rank* is selected, Spearman rank order correlation will be used to fit a correlation matrix for all columns. To select Rank use: `“CorrelationType”: “RANK”.`
		If *Copula* is selected, correlation will be fit using specified copulas. To select a Copula use: `“CorrelationType”: “COPULA”.`
		o If `“CorrelationType”: “COPULA”,` then copulas may be specified by setting the individual copula to true. If multiple copulas are selected, the first successfully fit copula will be used in the sample generation.
		RASON Decision Services offers 5 types of copulas:
		• STUDENT
		• CLAYTON
		• FRANK
		• GUMBEL
		• GAUSS

Generating Data Options		
randomGeneratorType	“randomGeneratorType”: <Type> Type = HDR, LECUYER_CMRG, MERSENNE_TWISTER, PARK_MILLER, WELL "randomGeneratorType": "HDR" "randomGeneratorType": "PARK_MILLER" "randomGeneratorType": "LECUYER_CMRG" "randomGeneratorType": "WELL" "randomGeneratorType": "MERSENNE_TWISTER"	Use this option to select a random number generation algorithm. RASON Decision Services includes an advanced set of random number generation capabilities. Computer-generated numbers are never truly “random,” since they are always computed by an algorithm – they are called *pseudorandom* numbers. A random number generator is designed to quickly generate sequences of numbers that are as close to being statistically independent as possible. Eventually, an algorithm will generate the same numbers seen sometime earlier in the sequence, and at this point the sequence will begin to repeat. The *period* of the random number generator is the number of values it can generate before repeating. A long period is desirable, but there is a tradeoff between the length of the period and the degree of statistical independence achieved within the period. Hence, RASON Decision Services offers a choice of five random number generators: o *Park-Miller* (PARK_MILLER)“Minimal” Generator with Bayes-Durham shuffle and safeguards. This generator has a period of 2^{31}-2. Its properties are good, but the following choices are usually better. o Combined Multiple Recursive Generator of L’Ecuyer (LECUYER_CMRG). This generator has a period of 2^{191}, and excellent statistical independence of samples within the period. o Well Equidistributed Long-period Linear (WELL) generator of Panneton, L’Ecuyer and Matsumoto. This generator combines a long period of 2^{1024} with very good statistical independence. o *Mersenne Twister* (default setting - MERSENNE_TWISTER) generator of Matsumoto and Nishimura. This generator has the longest period of 2^{19937}-1, but the samples are not as “equidistributed” as for the WELL and L-Ecuyer-CMRG generators.

		○ The *HDR* Random Number Generator (`HDR`), designed by Doug Hubbard. Permits data generation running on various computer platforms to generate identical or independent streams of random numbers.
randomSeed	`“randomSeed”: N,` where N is any positive integer `“randomSeed”: = 12345,`	Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the *same* sequence of random numbers is used for each simulation. When the seed is zero or RandomSeed is not specified, the random number generator is initialized from the system clock, so the sequence of random numbers will be different in each simulation. Set the seed to ensure that the results from one simulation to another are strictly comparable.

randomStream	`“RandomStreamType”: <type>`
	`Type = INDEPENDENT or SINGLE`
	`"randomStreamType": "SINGLE"`
	`"randomStreamType": "INDEPENDENT"`
		Use this option to select a *Single Stream* or an *Independent Stream (the default)* for each variable.
		If *Single Stream* is selected, a single sequence of random numbers is generated. Values are taken consecutively from this sequence to obtain samples for each selected variable. This introduces a subtle dependence between the samples for all distributions in one trial. In many applications, the effect is too small to make a difference – but in some cases, better results are obtained if *independent* random number sequences (streams) are used for each distribution in the model. RASON Decision Services offers this capability for Monte Carlo sampling and Latin Hypercube sampling; it does not apply to Sobol numbers.

samplingMethod	`“SamplingMethodType”: <type>,`
	`Type = MONTE_CARLO, LATIN_HYPERCUBE, SOBOL_RQMC`
	`"samplingMethodType": "MONTE_CARLO"`
	`"samplingMethodType": "LATIN_HYPERCUBE"`
	`"samplingMethodType": "SOBOL_RQMC`
		Use this option to select *Monte Carlo, Latin Hypercube*, or *Sobol RQMC* sampling.
		○ *Monte Carlo:* In standard Monte Carlo sampling, numbers generated by the chosen random number generator are used directly to obtain sample values. With this method, the variance or estimation error in computed samples is inversely proportional to the square root of the number of trials (controlled by the Sample Size); hence to cut the error in half, four times as many trials are required.

		RASON provides two other sampling methods than can significantly improve the 'coverage' of the sample space, and thus reduce the variance in computed samples. This means that you can achieve a given level of accuracy (low variance or error) with fewer trials. o *Latin Hypercube (default):* Latin Hypercube sampling begins with a stratified sample in each dimension (one for each selected variable), which constrains the random numbers drawn to lie in a set of subintervals from 0 to 1. Then these one-dimensional samples are combined and randomly permuted so that they 'cover' a unit hypercube in a stratified manner. o *Sobol RQMC (Randomized QMC)***.** Sobol numbers are an example of so-called "Quasi Monte Carlo" or "low-discrepancy numbers," which are generated with a goal of coverage of the sample space rather than "randomness" and statistical independence. A "random shift" is added to Sobol numbers, which improves their statistical independence.
sampleSize	`“sampleSize”: N,` where N is any positive integer `“sampleSize”: = 10`	Use this option to set the size of the generated sample. The default is 100.

Algorithm Parameters: Category Reduction for Transformation

Property	Example	Definition

`numCategories`	`"numCategories":[`
	`   [ 'X1', 3 ],`
	`   [ 'X3', 2 ]`
	`]`
		This utility helps you create a new categorical variable that reduces the number of categories. You can reduce the number of categories "by frequency" or "manually".

Actions ("actions")

Within the object "actions", user-defined attributes define action objects with the following properties: "data", "estimator", "model", "action", "parameters", "evaluations". An example of the action "nnpModel" (appearing in the Regression – NeuralNetwork.json RASON example on RASON.com) is shown below.

```
actions: {
      "nnpModel": {
            "trainData": 'myTrainData',
            "estimator": 'nnpEstimator',
            "action": "fit",
            "evaluations": [
                  "trainingLog",
                  "neuronWeights",
                  "numEpochsUsed",
                  "trainingTime",
                  "stoppingReason",
                  "partitionCausedStopping"
            ]
      },
}
```

In the code snippet above, the action, nnpModel, fits a model using the nnpEstimator to the "myTrainData" dataset. The results requested are: the training log (trainingLog), the neuron weights (neuronWeights), the number of epochs (numEpochsUsed), the solving time (trainingTime), the reason for stopping (stoppingReason) and the partition causing the stopping (partitionCausedStopping).

Evaluation results may either be 1. Part of the RASON response or 2. Bound to a writable datasource. In the example below, "regressionSummary" and "influenceDiagnostics" are part of the RASON response while "anova" and "detailedCoefficients" are bound to writable datasources, expANOVA and expDetCoeff, respectively.

```
"evaluations": [
 {"name": "anova",
  "binding": "expANOVA"
 },
  "regressionSummary",
  "influenceDiagnostics",
 {
"name":"detailedCoefficients",
  "binding": "expDetCoeff"
 }
]
```

The properties for the "actions" object are:

Property	Example	Definition
`data/trainData/validData`	`"trainData": "myTrainData"`	This property may be used interchangeably with the property, "data". In some algorithms, it is possible to provide both "trainData" and "validData" i.e. for classification and regression algorithms.
`estimator`	`"estimator": "nnpEstimator"`	Used to reference the estimator defined in the "estimator" stage. For more information on this stage, see the Estimator section above.
`action`	`"action":"fit"` **Parameter Options** • **forecast** • **fit** • **predict** • **transform**	Defines an "action" to be performed such as fit, predict, transform, or forecast. The first action, fit, fits the model given an estimator and training data. The remaining actions, predict, transform , and forecast, apply the

		fitted model to further operations on partitions or new data.
fittedModel	"fittedModel":"lrModel"	Used when scoring a model, this property is used to reference the model generated inside of the "model" object. For more information on scoring, see the example below.

parameters	parameters: {
	numForecasts: 7
	}
		The selection for this property depends on the "model" or "estimator" selected.
		Different values for scoring may be used when scoring multiple datasets using the same model.

evaluations	"evaluations": [
	{"name": "anova",
	"binding": "expANOVA"
	},
	"regressionSummary",
	"influenceDiagnostics",
	{
	"name":"detailedCoefficients",
	"binding": "expDetCoeff"
	}
	]
		The selection for this property depends on the "model" or "estimator" selected.
		In the example to the left, "regressionSummary" and "influenceDiagnostics" are part of the RASON response while "anova" and "detailedCoefficients" are bound to writable datasources, expANOVA and expDetCoeff, respectively.

The Model parameters described in the tables below are available for each corresponding algorithm.

Model Parameters: Association Rules

Parameter	Example	Definition
minConfidence	"minConfidence": 0.4	A value entered for this option specifies the minimum confidence threshold for rule generation. If A is the set of Antecedents and C the set of Consequents, then only those A =>C ("Antecedent implies Consequent") rules will qualify, for which the ratio (support of A U C) / (support of A) is greater than or equal to. The default setting is 50.

Model Parameters: Bagging – Classification and Regression

Parameter	Example	Definition
successClass	"successClass":"1"	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
successProbability	"successProbability":0.6	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If

		the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: Big Data – Sampler

There are no model parameters associated with this feature.

Model Parameters: Big Data – Summarizer

There are no model parameters associated with this feature.

Model Parameters: Binning

Parameters	Option Settings or Example	Explanation

`binValueOption`	`binValueOption: [`
	`[ 'x4', 'RANK' ],`
	`[ 'x2', 'MID_VALUE' ]`
	`]`
	Parameter Options
	• MEAN
	• MEDIAN
	• MID_VALUE
	• RANK
		When method = EQUAL_INTERVAL, use MID_VALUE to replace the value of the selected variable with the mid value of the interval for the assigned bin.
		When method = EQUAL_INTERVAL *OR EQUAL_COUNT*, use RANK to specify the *Start* value of the first bin and the *Interval* of each bin. Subsequent bin values will be calculated as the previous bin + interval value.
		When method = EQUAL_COUNT, use MEAN to replace the value of the selected variable with the mean of the interval for the assigned bin.
		When method = EQUAL_COUNT, use MEDIAN to replace the value of the selected variable value with the median of the interval for the assigned bin.

`rank`	`rank: [`
	`[ 'x4', 1.0, 5.0 ]`
	`]`
		This parameter is available when binValueOption is set to "RANK".
		Use the "rank" parameter to specify the *Start* value of the first bin and the *Interval* of each bin. Subsequent bin values will be calculated as the previous bin + interval value.

Model Parameters: Boosting – Classification and Regression

Parameter	Example	Definition

`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: Canonical Variate

`There are no model parameters associated with this feature.`

Model Parameters: Category Reduction for Transformation

Property	Example	Definition

`mapping`	`"mapping": [`
	`["Y","0",5],`
	`["Y","1",10]`
	`]`
		Assigns a specific category number to single or multiple categories.

Model Parameters: Decision Tree – Classification and Regression

Parameter	Example	Definition
`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5.

		This option is only supported when the number of classes is equal to 2.

Model Parameters: Factorization

There are no model parameters associated with this feature.

Model Parameters: Find Best Model – Classification and Regression

Parameter	Example	Definition

`bestLearnerMetric`	`"bestLearnerMetric": "ACCURACY"`
	Available classification metrics: "ACCURACY", "SPECIFICITY", "SENSITIVITY", "PRECISION" and "F1".
	Available regression metrics: "SSE", "MSE", "RMSE", "MAD" and "R2.
	
	`"trainScore": {`
	`  "data": "myTrainData",`
	`  "fittedModel": "fbmModel",`
	`  "parameters": {`
	`    `**`"bestLearnerMetric"`**`:`
	`      `**`"ACCURACY"`**
	`  },`
	`  "action": "predict",`
	`  "evaluations":[`
	`    `**`"modelPerformance"`**`,`
	`    `**`"bestLearner"`**`,`
	`    "prediction"`
	`  ]`
	`}`
		Use this parameter to allow the Find Best Model method to select the learner, from all available learners, that fits the best model to the dataset according to the selected metric.
		Two evaluators are available for this metric: "modelPerformance" and "bestLearner". *You'll need to add the evaluators, "modelPerformance" and "bestLearner" as "evaluations" to add them to the output.*
		The model performance table ("modelPerformance") contains fitting information pertaining to how well the available learners were able to fit a model to the dataset according to the selected metric.
		The best learner ("bestLearner) gives the name of the best learner for a given "predict" action. Note that the best learner may not be the learner used to actually score the data in the given action, see learnerForScoring in the output for more information on what learner was selected to perform scoring.

`useForScoring`	`"useForScoring"=true`	Enter this metric for *one* "predict" action. This parameter instructs the Find Best Model method to use the best learner found in the given action to score the data in *all* "predict" actions. 1. If no "predict' action contains "useForScoring"=true, then the best learner from the first "predict" action containing the "bestLearnerMetric" is selected to score *all* "predict" actions. 2. If multiple "predict" actions contain "useForScoring" = true, then the best learner from the first listed action is used to score *all* "predict" actions.

Model Parameters: Univariate Feature Selection

Parameters	Option Settings or Example	Explanation

numTopFeatures	"numTopFeatures" : 2	Model option only. Enter a value ranging from 1 to the number of features in the model. This value, along with the "usePvalueForSelection" option setting, will be used to determine the variables included in the *Top Features Table* and *Feature Importance Plot*. This option has a default setting of "2".
usePvalueForSelection	"usePvalueForSelection":"true"	Model option only. If *"True"*, then the variables will be ranked from smallest to largest using the P-value of the measure or statistic selected.

Model Parameters: Logistic/Linear Wrapping Feature Selection

Parameters	Option Settings or Example	Explanation
numTopFeatures	"numTopFeatures" : 2	Enter a value ranging from 1 to the number of features in the model.

Model Parameters: Hierarchical Clustering

numClusters	"numClusters" : 10	The agglomerative method of hierarchical clustering continues to form clusters until only one cluster is left. This option lets you stop the process at a given number of clusters.
numDendrogramLeaves	"numDendrogramLeaves" : 10	Use this option to define the maximum number of leaves in the dendrogram tree.

Model Parameters: Imputation

imputation	"imputation": [["F", "medium"]]	Use the imputation model parameter to set the user defined value.

Model Parameters: k-Means Clustering

There are no model parameters associated with this feature.

Model Parameters: Discriminant Analysis – Classification

Parameter	Example	Definition
successClass	"successClass":"1"	**For classification models only.** Select the class to be considered a "success" or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
successProbability	"successProbability":0.6	**For classification models only.**

		Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: Linear/Logistic – Classification and Regression

Parameter	Example	Definition
`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: Latent Semantic Analysis

There are no model parameters associated with this feature.

Model Parameters: Naïve Bayes – Classification

Parameter	Example	Definition
`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If

		the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: k-Nearest Neighbors – Classification and Regression

Parameter	Example	Definition
`includeTies`	`"includeTies": true`	If includeTies = True, all points with distance equal to kth nearest neighbor are included in the result. If includeTies = False, exactly k nearest neighbors are returned.
`numNeighbors`	`"numNeighbors":3`	This is the parameter k in the k-nearest neighbor algorithm.
`stable`	`"stable":true`	If stable = true, the tied neighbors (up to kth neighbor) remain in the original order. If stable = false, the tied neighbors (up to kth neighbor) are in pseudo-random order.
`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

`weightingScheme`	`"weightingScheme":`
	`"INVERSE_DISTANCE"`
	`Parameter Options`
	• `EQUAL`
	• `INVERSE_DISTANCE`
		Use this option to select the weighting scheme: equal or inverse distance.

Model Parameters: Neural Network – Classification and Regression

Parameter	Example	Definition
`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a “success” or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a “success” (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a “non-success” (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: One Hot Encoder

`There are no model parameters associated with this feature.`

Model Parameters: Partitioning

Property	Example	Definition
`partition`	`"partition": "Training"`	Specifies the partition to transform.

Model Parameters: PCA

Parameters	Option Settings or Example	Explanation
`numPrincipalComponents`	`"numPrincipalComponents": 2`	This option is mutually exclusive with varianceCutoff. Use either numPrincipalComponents to select the number of principal components displayed in the output or varianceCutoff, but not both. This option specifies a fixed number of components. Enter an integer value from 1 to n where n is the number of *Input variables* in the model.

`varianceCutoff`	`"varianceCutoff" : 0.98`	This option is mutually exclusive with numPrincipalComponents. Use either numPrincipalComponents to select the number of principal components displayed in the output or varianceCutoff, but not both. Use this option to calculate the minimum number of principal components required to account for the percentage of variance entered for this option.

Model Parameters: Random Trees -- Classification and Regression

Parameter	Example	Definition
`successClass`	`"successClass":"1"`	**For classification models only.** Select the class to be considered a "success" or the significant class. This option is only supported when the number of classes in the output variable is equal to 2.
`successProbability`	`"successProbability":0.6`	**For classification models only.** Enter a value between 0 and 1 for this option to denote the cutoff probability for success. If the calculated probability for success for an observation is greater than or equal to this value, than a "success" (or a 1) will be predicted for that observation. If the calculated probability for success for an observation is less than this value, then a "non-success" (or a 0) will be predicted for that observation. The default value is 0.5. This option is only supported when the number of classes is equal to 2.

Model Parameters: Sampling/Stratified Sampling

There are no model parameters associated with this feature.

Model Parameters: Rescaler

There are no model parameters associated with this feature.

Model Parameters: Smoothing Methods: Double Exponential, Exponential, Holt Winters Additive, Holt Winters Multiplicative, Holt Winters No Trend, Moving Average

Parameter	Example	Definition
`confidenceLevel`	`"confidenceLevel":0.9`	Sets the desired confidence level here. (The default level is 95%.) The Lower and Upper values of the computed confidence levels will be included in the output. The forecasted

		value will be guaranteed to fall within this range for the specified confidence level.
`numForecasts`	`"numForecasts": 4`	Sets the number of forecasts.

Model Parameters: TFIDF

Parameters	Option Settings or Example	Explanation
`weightingSchemeDocument`	`"weightingSchemeDocument": "INVERSE"`	See explanation below
`weightingSchemeNormalization`	`"weightingSchemeNormalization": "NONE"`	See explanation below.
`weightingSchemeTerm`	`"weightingSchemeTerm": "LOGARITHMIC",`	See explanation below.

Explanation: Using these three options, users can select their own choices for local weighting, global weighting, and normalization. Please see the table below for definitions regarding options for Term Frequency, Document Frequency and Normalization.

Local Weighting		Global Weighting		Normalization	
Binary	$lw_{td} = \begin{cases} 1, \text{if } tf_{td} > 0 \\ 0, \text{if } tf_{td} = 0 \end{cases}$	None	$gw_t = 1$	None	$n_d = 1$
Raw Frequency	$lw_{td} = tf_{td}$	Inverse	$gw_t = \log_2 \frac{N}{1 + df_t}$	Cosine	$n_d = \frac{1}{\lVert \overline{g_d} \rVert_2}$
Logarithmic	$lw_{td} = \log(1 + tf_{td})$	Normal	$gw_t = \frac{1}{\sqrt{\sum_d tf_{td}^2}}$		
Augnorm	$lw_{td} = \frac{\left(\frac{tf_{td}}{\max\limits_t tf_{td}}\right) + 1}{2}$	GF-IDF	$gw_t = \frac{cf_t}{df_t}$		
		Entropy	$gw_t = 1 + \sum_d \frac{p_{td} \log p_{td}}{\log N}$		
		IDF probability	$gw_t = \log_2 \frac{N}{1 + df_t}$		

Notations:

- tf_{td} **– frequency of term** t **in a document** d**;**
- df_t **– document frequency of term** t**;**
- lw_{td} **– local weighting of term** t **in a document** d**;**
- gw_{td} **– global weighting of term** t **in a document** d**;**
- n_d **– normalization of vector of terms representing the document** d**;**
- N **– total number of documents in the collection;**

- **cf_t – collection frequency of term t;**
- **p_{td} – estimated probability of term t to appear in a document d $\left(p_{td} = {}^{tf_{td}}/_{cf_t}\right)$;**
- **$\overline{g_d}$ – vector of terms representing the document d.**

Finally, the element T_{td} of Term-Document Matrix is computed as $T_{td} = lw_{td} * gw_t * n_d, \forall t, d$

Model Parameters: Time Series

Parameter	Example	Definition
`confidenceLevel`	`"confidenceLevel":0.9`	Sets the desired confidence level here. (The default level is 95%.) The Lower and Upper values of the computed confidence levels will be included in the output. The forecasted value will be guaranteed to fall within this range for the specified confidence level.
`numForecasts`	`"numForecasts": 4`	Sets the number of forecasts.

The "evaluations", or quantities to be computed and reported back, described in the tables below are available for each corresponding algorithm.

Evaluations Common to All Rason DM Methods and Algorithms

Evaluation	Action	Definition
`fittedModeljson`	Fit	Returns the data mining model in JSON format.

Evaluations: Big Data Common Evaluations

Evaluation	Action	Definition
`clusterInfo`	Transform	Returns the Apache Spark REST Server URL.
`durationInfo`	Transform	Returns the elapsed time since job submission if the job is still RUNNING and the cluster total compute time if the job is FINISHED.
`jobID`	Transform	Returns the ID from a previous submission. Use this evaluation when submitting a sampling job, not for obtaining results.
`schema`	Transform	Returns the full and sampled data schema.
`solverDatasets`	Transform	Lists the preloaded datasets on Frontline Systems cluster.
`transformation`	Transform	Returns the final transformation (transformed dataset).

Evaluations: Binning

Evaluation	Action	Definition
`breakPoints`	Fit	Returns the breakpoint (largest value) for each bin.
`frequencyTable`	Transform	Returns the frequency table information including the the lower and upper values for each bin and the records assigned to each bin.
`numBins`	Fit	Returns the number of bins.
`transformation`	Transform	Returns the final transformation (transformed dataset).

Evaluations: Canonical Variates

Evaluation	Action	Definition
`canonicalVariates`	Fit/Transform	Returns the canonical variates for the data based on an orthogonal representation of the original variates.
`transformation`	Transform	Returns the final transformation (transformed dataset).

Evaluations: Classification – Common Parameters

Evaluation	Action	Definition
`accuracy`	Predict	Returns the accuracy metric (# correct)
`auc`	Predict	Returns the AUC for the ROC Curve
`confusionMatrix`	Predict	Returns the confusion matrix for a classification method.
`decileChart`	Predict	Returns the decile chart for a classification method.
`f1`	Predict	Returns the F1 Score.
`liftChart`	Predict	Returns the lift chart information for a classification method.
`metrics`	Predict	Returns the following metrics for a classification model: accuracy, specificity, sensitivity, precision, the F1 score, and AUC.
`posteriorProbability`	Predict	Returns the posterior probability
`precision`	Predict	Returns the precision matrix. Precision is the probability of correctly identifying a randomly selected record as one belonging to the Success class. TP/(TP + FP)
`prediction`	Predict	Returns the prediction label for each record in the dataset.
`recall`	Predict	Returns the Recall (Sensitivity) metric. Recall (or Sensitivity) measures the percentage of

		actual positives which are correctly identified as positive. TP/(TP+FN)
`rocCurve`	Predict	Returns the ROC Curve information
`sensitivity`	Predict	Returns the sensitivity metric.
`specificity`	Predict	Returns the Specificity metric. Specificity (SPC) or True Negative Rate =TN / (FP + TN)

Evaluations: Classification – Decision Trees

Evaluation	Action	Definition
`categoricalFeaturesInfo`	Fit	Returns information on the categorical features included in the model.
`featureImportance`	Fit	Returns the variables that are included in the model along with their Importance value.
`pruningLog`	Fit	Returns the prune log.
`trainingLog`	Fit	Returns the training log.
`treeDiagram`	Fit	Returns the tree diagram.
`treeRules`	Fit	Returns the tree rules.

Evaluations: Classification – Ensemble Methods Common Options

Evaluation	Action	Definition
`categoricalFeaturesInfo`	Fit	Returns information on the categorical features included in the model.
`numWeakLearners`	Fit	Returns the number of weak learners
`weakLearnerModels`	Fit	Returns the weak learner models.

Evaluations: Classification -- Discriminant Analysis

Evaluation	Action	Definition
`linearDiscriminantFunctions`	Fit	Returns the Linear Discriminant Functions table. In this table, there will be a function for each class. Each variable will be assigned to the class that contains the higher function value.
`quadraticDiscriminantFunctions`		Returns the Quadratic Discriminant Functions tables. One table per class will be returned in the results, i.e. if there are two classes in the model, two quadratic functions tables will be returned. Each variable will be assigned to the class that contains the higher function value.

Evaluations: Classification – Logistic Regression

Evaluation	Action	Definition
`categoricalFeaturesInfo`	Fit	Returns information on the categorical features included in the model.
`coefficients`	Fit	Returns the coefficient estimates.
`detailedCoefficients`	Fit	Returns the coefficient estimate, the standard error of the coefficient, the p-value, the odds ratio for each variable (which is simply e^x where x is the value of the coefficient) and confidence interval for the odds. (Note for the Intercept term, the Odds Ratio is calculated as exp^0.)
`entranceTolerance`	Fit	Returns the tolerance threshold. All predictors eligible to enter the model must pass this threshold.
`multicollinearityDiagnostics`	Fit	Returns Collinearity Diagnostics which help assess whether two or more variables so closely track one another as to provide essentially the same information. The columns represent the variance components (related to principal components in multivariate analysis), while the rows represent the variance proportion decomposition explained by each variable in the model. The eigenvalues are those associated with the singular value decomposition of the variance-covariance matrix of the coefficients, while the condition numbers are the ratios of the square root of the largest eigenvalue to all the rest. In general, multicollinearity is likely to be a problem with a high condition number (more than 20 or 30), and high variance decomposition proportions (say more than 0.5) for two or more variables.
`multipleR2`	Fit	
`numIterations`	Fit	
`predictorScreeningInfo`	Fit	A preprocessing feature selection step is included to take advantage of automatic variable screening and elimination using Rank-Revealing QR Decomposition. This allows the identification of variables causing multicollinearity, rank deficiencies and otherproblems that would otherwise cause the algorithm to fail. Information about “bad” variables is used in Variable Selection and Multicollinearity Diagnostics and in computing other reported statistics. Included and excluded predictors are returned for this command. All predictors must meet the tolerance threshold to be eligible to enter the model. This denotes a tolerance beyond

		which a variance – covariance matrix is not exactly singular to within machine precision. The test is based on the diagonal elements of the triangular factor R resulting from Rank-Revealing QR Decomposition. Predictors that do not pass the test are excluded. Note: If a predictor is excluded, the corresponding coefficient estimates will be 0 in the regression model and the variable – covariance matrix will contain all zeros in the rows and columns that correspond to the excluded predictor. Multicollinearity diagnostics, variable selection and other remaining output will be calculated for the reduced model. The design matrix may be rank-deficient for several reasons. The most common cause of an ill-conditioned regression problem is the presence of feature(s) that can be exactly or approximately represented by a linear combination of other feature(s). For example, assume that among predictors you have 3 input variables X, Y, and Z where Z = a * X + b * Y where a and b are constants. This will cause the design matrix to not have a full rank. Therefore, one of these 3 variables will not pass the threshold for entrance and will be excluded from the final regression model.
`regressionSummary`	Fit	Returns the the residual degrees of freedom (#observations - #predictors), a standard deviation type measure for the model (which typically has a chi-square distribution), the number of iterations required to fit the model, and the Multiple R-squared value. The multiple R-squared is the r-squared value for a logistic regression model , defined as - $R^2 = (D_0\text{-}D)/D_0$, where D is the Deviance based on the fitted model and D_0 is the deviance based on the null model. The null model is defined as the model containing no predictor variables apart from the constant.
`residuals`	Fit	
`residualDeviance`	Fit	
`residualDF`	Fit	
`varianceCovariance`	Fit	Returns the Variance – Covariance matrix.

Evaluations: Classification – Naïve Bayes

Evaluation	Action	Definition
`classFrequency`	Fit	Returns the total predicted number of cases assigned to each class.

`dataFrequency`	Fit	Returns the Prior Conditional Probabilities by feature (column) for the Training dataset.
`logDensity`	Predict	Returns the Log Densities for each partition. Log PDF, or Logarithm of Unconditional Probability Density, is the distribution of the predictors marginalized over the classes and is computed using: $\log[P\{X_1,\ldots,X_n\}] = \log\left[\sum_{c=1}^{C} P\{X_1,\ldots,X_n,Y = c\}\right] = \log\left[\sum_{c=1}^{C} \pi\{\,y = c\}\,P\{X_1,\ldots,X_n \mid Y = c\}\right]$ where$\pi\{Y = c\}$ is a prior class probability
`priorConditionalProbability`	Fit	Returns the Prior Conditional Probabilities for each case by variable.

Evaluations: Classification – Neural Networks

Evaluation	Action	Definition
`categoricalFeaturesInfo`	Fit	Returns information on the categorical features included in the model.
`neuronWeights`	Fit	Returns the interlayer connections' weights table.
`numEpochsUsed`	Fit	Returns the number of epochs performed.
`partitionCausedStopping`	Fit	Returns the partition used for error computation.
`stoppingReason`	Fit	Returns the reason for stopping.
`trainingLog`	Fit	Returns the neural network training log.
`trainingTime`	Fit	Returns the time taken to train the network.

Evaluations: Classification – Random Trees

Evaluation	Action	Definition
`categoricalFeaturesInfo`	Fit	Returns information on the categorical features included in the model.
`featureImportance`	Fit	Returns the variables that are included in the model along with their Importance value.
`numWeakLearners`	Fit	Returns the number of weak learners
`weakLearnerModels`	Fit	Returns the weak learner models.

Evaluations: Clustering – Hierarchical

Evaluation	Action	Definition
dendrogram	Predict	Returns the dendrogram information
dendrogramChart	Predict	Returns the dendrogram information in chart format.
mergingHistory	Fit	Returns the history of the cluster formation. Initially, each individual case is considered its own cluster (single member in each cluster) beginning with # clusters = # cases.
prediction	Predict	Returns the predicted values
subclusterLegend	Predict	Returns the subclusters.

Evaluations: Clustering – kMeans

Evaluation	Action	Definition
clusterCenters	Fit	Displays detailed information about the clusters formed by the k-Means Clustering algorithm: the final centroids. If the input data was normalized, information is displayed in original and normalized coordinates.
clustersSummary	Predict	Displays the number of records (observations) included in each cluster and the within-cluster average distance. This information can be used to better understand the data partitioning: how large and how sparse the resulting clusters are.
interclusterDistances	Fit	Returns the inter-cluster distances.
prediction	Predict	Returns the predicted values.
randomCentersSummary	Fit	Returns the information about the initial search for the best centroid assignment. The assignment marked by “Best Start” is used as the initial assignment of the centroids.
recordToClusterDistance	Predict	Returns the distance between each record and it's assigned cluster.

Evaluations: Factoring

Evaluation	Action	Definition
frequencyTable	Transform	Returns the frequency table information for the factored variable(s).
transformation	Transform	Returns the final transformation (transformed dataset).

Evaluations: Feature Selection - Linear/Logistic Wrapping

Evaluation		Definition

`bestSubsets`	Fit	Returns the best subsets as determined by various error values and the probability. Use "bestSubsetsDetails" to view these error values and probability.
`bestSubsetsDetails`	Fit	Returns the number of predictors, the residual sum of squares (RSS), Mallows CP, and the Probability for each subset. RSS is the residual sum of squares, or the sum of squared deviations between the predicted probability of success and the actual value (1 or 0). "Mallows Cp" is a measure of the error in the best subset model, relative to the error incorporating all variables. Adequate models are those for which Cp is roughly equal to the number of parameters in the model (including the constant), and/or Cp is at a minimum. "Probability" is a quasi hypothesis test of the proposition that a given subset is acceptable; if Probability < .05 we can rule out that subset.
`transformation`	Transform	Returns the final transformation (transformed dataset).

Evaluations: Feature Selection - Univariate

Evaluation	Action	Definition
`fsPlot`	Transform	Plots the top most important or relevant features as determined by the value entered for "numTopFeatures" parameter setting.
`statistics`	Fit	Returns the value for the requested statistic (as set using the "metric" estimator parameter).
`topFeaturesInfo`	Transform	Produces a table containing the top number of features as determined by "numTopFeatures" parameter setting.
`transformation`	Transform	Returns the final transformation (transformed dataset).

Evaluations: Find Best Model – Classification and Regression

Evaluation	Action	Definition

`bestLearner`	Predict
	`"trainScore": {`
	`  "data": "myTrainData",`
	`  "fittedModel": "fbmModel",`
	`  "parameters": {`
	`    "bestLearnerMetric":`
	`    "ACCURACY"`
	`  },`
	`  "action": "predict",`
		This evaluation is available when the "bestLearnerMetric" parameter is present in the "predict" action.
		This evaluation returns the best learner, for the given "predict" action, according to the "bestLearnerMetric".

	"evaluations": ["modelPerformance", **"bestLearner"**, "learnerForScoring", "prediction"] },	

learnerForScoring	Predict
	"trainScore": {
	"data": "myTrainData",
	"fittedModel": "fbmModel",
	"parameters": {
	"bestLearnerMetric":
	"ACCURACY",
	},
	"action": "predict",
	"evaluations": [
	"modelPerformance",
	"bestLearner",
	"learnerForScoring",
	"prediction"
	]
	},
		This evaluation returns the learner used to score all "action": "predict" methods in the RASON model. All "predict" methods will contain this output.

Messages	Fit	The Find Best Model "fit" action has only one possible evaluation, "messages". This evaluation reports the fitting log which is where any warnings/failures that occur during the fitting process will be reported.

modelPerformance	Predict
	"trainScore": {
	"data": "myTrainData",
	"fittedModel": "fbmModel",
	"parameters": {
	"bestLearnerMetric":
	"ACCURACY"
	},
	"action": "predict",
	"evaluations": [
	"modelPerformance",
	"bestLearner",
	"learnerForScoring",
	"prediction"
	]
	},
		This evaluation is available when the "bestLearnerMetric" parameter is present in the "predict" action.
		This evaluation returns a table containing the model performance metrics for the given "predict" action, for each available learner in the Find Best Model method.
		For classification models, the returned metrics are: Accuracy, Specificity, Sensitivity, Precision and F1 score.
		For regression models, the returned metrics are: SSE, MSE, RMSE, MAD and R2.

Evaluations: Forecasts – Common Parameters

Evaluation	Action	Definition
Cfe	Forecast, Transform	Returns the cumulative forecast error
coefficientsInfo	Fit	Returns the coefficient for each term in the ARIMA model.

forecast	Forecast	Returns the forecasted values.
mad	Forecast, Transform	Returns MAD (Mean Absolute Deviation).
mfe	Forecast, Transform	Returns the MFE (Mean Forecast Error).
mape	Forecast, Transform	Returns the Mean Absolute Percentage Error (MAPE).
metrics	Forecast	Returns the following metrics: SSE, MSE, MAPE, MAD, CFE, MFE and TSE.
mse	Transform	Returns MSE (Mean Squared Error).
residuals	Transform	Returns the residuals calculated by subtracting the predicted value by the actual value.
sse	Forecast, Transform	Returns the Sum of Squared Error (SSE).
transformation	Transform	Returns the final transformation (transformed dataset).
tse	Forecast, Transform	Returns the Tacking Signal Error.
tsPlot	Forecast, Transform	Returns the time series plot: Actual vs Forecast.

Evaluations: Regression – Common Parameters

Evaluation	Action	Definition
aoc	Predict	Returns the area Over the Curve (AOC) in an RROC Curve.
decileChart	Predict	Retuns the decilewise lift curve which is drawn as the decile number versus the cumulative actual output variable value divided by the decile's mean output variable value. The bars in this chart indicate the factor by which the predicted model outperforms a random assignment, one decile at a time.
liftChart	Predict	Returns information for both Original and Alternative Lift Charts.
prediction	Predict	Returns predicted values.
mad	Predict	Returns MAD (Mean Absolute Deviation).
metrics	Predict	Returns various metrics such as MSE, R2, RMSE, etc.
mse	Predict	Returns MSE (Mean Squared Error).
r2	Predict	Returns Coefficient of Determination (R^2)
residuals	Predict	Returns the residuals calculated by subtracting the predicted value by the actual value.
rmse	Predict	Returns Root Mean Squared Error (RMSE).
rrocCurve	Predict	Returns RROC Curve information.
ss	Predict	Returns the Sum of Squares (SS).

sse	Predict	Returns the Sum of Squared Error (SSE).
sst	Predict	Returns the Sum of Squares Total (SST).

Evaluations: Regression – Linear Regression

Evaluation	Action	Definition
anova	Fit	Returns the Analysis of Variance (ANOVA)
coefficients	Fit	Returns the variable coefficients
detailedCoefficients	Fit	Returns the coefficient estimate, the standard error of the coefficient, the T-statistic, the p-value, and confidence interval.
detailedResiduals	Fit	Returns the raw, standardized, studentized and deleted residuals.
entranceTolerance	Fit	Returns the tolerance threshold. All predictors eligible to enter the model must pass this threshold.
influenceDiagnostics	Fit	Returns Cook's Distance, DFFits, Covariance ratio, Leverage, and Delete-1 Variance metrics
multicollinearityDiagnostics	Fit	Returns Collinearity Diagnostics which help assess whether two or more variables so closely track one another as to provide essentially the same information. The columns represent the variance components (related to principal components in multivariate analysis), while the rows represent the variance proportion decomposition explained by each variable in the model. The eigenvalues are those associated with the singular value decomposition of the variance-covariance matrix of the coefficients, while the condition numbers are the ratios of the square root of the largest eigenvalue to all the rest. In general, multicollinearity is likely to be a problem with a high condition number (more than 20 or 30), and high variance decomposition proportions (say more than 0.5) for two or more variables.
predictorScreeningInfo	Fit	A preprocessing feature selection step is included to take advantage of automatic variable screening and elimination using Rank-Revealing QR Decomposition. This allows the identification of variables causing multicollinearity, rank deficiencies and otherproblems that would otherwise cause the algorithm to

		fail. Information about “bad” variables is used in Variable Selection and Multicollinearity Diagnostics and in computing other reported statistics. Included and excluded predictors are returned for this command. All predictors must meet the tolerance threshold to be eligible to enter the model. This denotes a tolerance beyond which a variance – covariance matrix is not exactly singular to within machine precision. The test is based on the diagonal elements of the triangular factor R resulting from Rank-Revealing QR Decomposition. Predictors that do not pass the test are excluded. Note: If a predictor is excluded, the corresponding coefficient estimates will be 0 in the regression model and the variable – covariance matrix will contain all zeros in the rows and columns that correspond to the excluded predictor. Multicollinearity diagnostics, variable selection and other remaining output will be calculated for the reduced model. The design matrix may be rank-deficient for several reasons. The most common cause of an ill-conditioned regression problem is the presence of feature(s) that can be exactly or approximately represented by a linear combination of other feature(s). For example, assume that among predictors you have 3 input variables X, Y, and Z where Z = a * X + b * Y where a and b are constants. This will cause the design matrix to not have a full rank. Therefore, one of these 3 variables will not pass the threshold for entrance and will be excluded from the final regression model.
`newIntervals`	Predict	Returns the lower and upper confidence and prediction intervals for the predicted values.
`regressionSummary`	Fit	Returns the the residual degrees of freedom (#observations - #predictors), a standard deviation type measure for the model (which typically has a chi-square distribution), the number of iterations required to fit the model, and the Multiple R-squared value. The multiple R-squared is the r-squared value for a logistic regression model , defined as - $R^2 = (D_0\text{-}D)/D_0$, where D is the Deviance based on the fitted model

		and D_0 is the deviance based on the null model. The null model is defined as the model containing no predictor variables apart from the constant.
`trainingIntervals`	Fit	Returns the lower and upper confidence and prediction intervals for the training partition.
`varianceCovariance`	Fit	Returns the variance-covariance matrix.

Evaluations: SyntheticDataGenerator

Evaluation	Action	Definition
Synthetic Data Generation		
bestMetalog	`"bestMetalog"`	Use BestMetalog to obtain the number of terms for the best fitted Metalog distribution for each column.
metalogCoefficients	`"metalogCoefficients"`	Obtains the coefficients for fitted Metalog Distributions for all or specific columns.
metalogFitted	`"metalogFitted"`	Checks whether there was at least one feasible Metalog distribution fitted for all or specific columns.
metalogGOF	`"MetalogGOF"`	Gets the detailed report of Goodness of Fit tests for fitted Metalog Distributions for all or specific columns.
Correlation Evaluations		
correlationFitted	`"correlationFitted"`	Obtains information on the correlation fitting. `"selectedCopula"` – Determines what copula was selected if copua fitting was requested. `"correlationSigma"` – Obtains the correlation matrix , when applicable – i.e. if rank correlation or Gauss/Student Copula was used. `"copulaTheta"` – Use this result to obtain the fitted correlation theta value when selected copula is Clayton, Frank or Grumbel. `"copulaDF"` – Use this result to obtain the degrees of freedom when copula is Student. Example code above first checks if the correlations have been fitted, and then, depending on the correlation type, (rank or copula), reports applicable results.

Evaluations: Summarizer

Summary	`“summary”`	All statistics generated using the Summarizer ealuations are briefly described below. Mean, the average of all the values. Standard Deviation, the square root of variance. Variance, the spread of the distribution of values. Skewness, which describes the *asymmetry* of the distribution of values. Kurtosis, which describes the *peakedness* of the distribution of values. Mode, the most frequently occurring single value. Minimum, the minimum value attained. Maximum, the maximum value attained. Range, the difference between the maximum and minimum values.
advancedSummary	`“advancedSummary”`	Advanced Summary Mean Abs. Deviation, returns the average of the absolute deviations. SemiVariance, measure of the dispersion of values. SemiDeviation, *one-sided* measure of dispersion of values.

		Value at Risk 95%, the maximum loss that can occur at a given confidence level. Cond. Value at Risk, is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs. Mean Confidence, returns the confidence “half-interval” for the estimated mean value (returned by the PsiMean() function. Std. Dev. Confidence 95%, returns the confidence ‘half-interval’ for the estimated standard deviation of the simulation trials (returned by the PsiStdDev() function). Coefficient of Variation, is defined as the ratio of the standard deviation to the mean. Standard Error, defined as the standard deviation of the sample mean. Expected Loss, returns the average of all negative data multiplied by the percentrank of 0 among all data. Expected Loss Ratio, returns the expected loss ratio. Expected Gain returns the average of all positive data multiplied by 1 - percentrank of 0 among all data. Expected Gain Ratio, returns the expected gain ratio. Expected Value Margin, returns the expected value margin
percentiles	`“percentiles”`	Generates numeric percentile values (from 1% to 99%) computed using all values for the variable. For example, the 75th Percentile value is a number such that three-quarters of the values occurring in the last simulation are less than or equal to this value.
sixSigma	`“sixSigma”`	Generates various computed Six Sigma measures, described below. These functions compute values related to the Six Sigma indices used in manufacturing and process control. **SigmaCP**: SixSigmaCP (cell,lower_limit,upper_limit) A Six Sigma index, SixSigmaCP predicts what the process is capable of producing if the process mean is centered between the lower and upper limits. This index assumes the process output is normally distributed. $Cp = \frac{UpperSpecificationLimit - LowerSpecificationLimit}{6\hat{\sigma}}$ where $\hat{\sigma}$ is the estimated standard deviation of the process. **SigmaCPK:** SixSigmaCPK (cell,lower_limit,upper_limit) A Six Sigma index, SixSigmaCPK predicts what the process is capable of producing if the process mean is not centered between the lower and upper limits. This

		index assumes the process output is normally distributed and will be negative if the process mean falls outside of the lower and upper specification limits. $Cpk = \frac{MIN(UpperSpecificationLimit-\hat{\mu},\hat{\mu}-LowerSpecificationLimit)}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process. **SigmaCPKLower:** SixSigmaCPKLower(cell,lower_limit) A Six Sigma index, SixSigmaCPKLower calculates the one-sided Process Capability Index based on the lower specification limit. This index assumes the process output is normally distributed. $Cp, lower = \frac{\hat{\mu}-LowerSpecificationLimit}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process. **SigmaCPKUpper:** SixSigmaCPKUpper(cell,upper_limit) A Six Sigma index, SixSigmaCPKUpper calculates the one-sided Process Capability Index based on the upper specification limit. This index assumes the process output is normally distributed. $Cp, upper = \frac{UpperSpecificationLimit-\hat{\mu}}{3\hat{\sigma}}$ where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process. **SigmaCPM:** SixSigmaCPM(cell,lower_limit,upper_limit,target) A Six Sigma index, SixSigmaCPM calculates the capability of the process around a target value. This index is referred to as the Taguchi Capability Index. This index assumes the process output is normally distributed and is always positive. $Cpm = \frac{\hat{C}p}{\sqrt{1+(\frac{\hat{\mu}-T}{\hat{\sigma}})^2}}$ where $\hat{C}p$ is the process capability (SigmaCP), $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and T is the target process mean. **SigmaDefectPPM**: SixSigmaDefectPPM(cell,lower_limit,upper_limit) A Six Sigma index, SixSigmaDefectPPM calculates the Defective Parts per Million. $DPMO = (\delta^{-1}(\frac{LowerSpecificationLimit-\hat{\mu}}{\hat{\sigma}}) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}})) * 1000000$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

		SigmaDefectShiftPPM: SixSigmaDefectPPM(cell,lower_limit,upper_limit) A Six Sigma index, SixSigmaDefectShiftPPM calculates the Defective Parts per Million with an added shift. $DPMOShift = (\delta^{-1}(\frac{LowerSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift)) * 1000000$ where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaDefectShiftPPMLower:** SixSigmaDefectShiftPPMLower(cell,lower_limit,shift) A Six Sigma index, SixSigmaDefectShiftPPMLower calculates the Defective Parts per Million, with a shift, below the lower specification limit. $DPMOshift,lower = (\delta^{-1}(\frac{LowerSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$ Where $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaDefectShiftPPMUpper:** SixSigmaCPKUpper(cell,upper_limit) A Six Sigma index, SixSigmaDefectShiftPPMUpper calculates the Defective Parts per Million, with a shift, above the lower specification limit. $DPMOshift,upper = (\delta^{-1}(\frac{UpperSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$ where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaK: SixSigmaK(cell,lower_limit,upper_limit)** A Six Sigma index, SixSigmaK calculates the Measure of Process Center and is defined as: $1 - \frac{2*MIN(UpperSpecificationLimit-\hat{\mu},\hat{\mu}-LowerSpecificationLimit)}{UpperSpecificationLimit-LowerSpecificationLimit}$ where$\hat{\mu}$ is the process mean. **SigmaLowerBound: SixSigmaLowerBound(cell,number_stdev)** A Six Sigma index, SixSigmaLowerBound calculates the Lower Bound as a specific number of standard deviations below the mean and is defined as: $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

		SigmaProbDefectShift: SixSigmaProbDefectShift(cell,lower_limit, upper_limit,shift) A Six Sigma index, SixSigmaProbDefectShift calculates the Probability of Defect, with a shift, outside of the upper and lower limits. This statistic is defined as: $\delta^{-1}\left(\frac{LowerSpecificationLimit\ -\hat{\mu}}{\hat{\sigma}} - Shift\right) +$ $1 - \delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift)$ where $\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaProbDefectShiftLower: SixSigmaProbDefectShiftLower(cell,lower_limit, shift)** A Six Sigma index, SixSigmaProbDefectShiftLower calculates the Probability of Defect, with a shift, outside of the lower limit. This statistic is defined as: $\delta^{-1}(\frac{LowerSpecificationLimit\ -\hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaProbDefectShiftUpper: SixSigmaProbDefectShiftUpper(cell,upper_limit, shift)** A Six Sigma index, SixSigmaProbDefectShiftUpper calculates the Probability of Defect, with a shift, outside of the upper limit. This statistic is defined as: $1 - \delta^{-1}(\frac{UpperSpecificationLimit\ -\hat{\mu}}{\hat{\sigma}} - Shift)$ where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaSigmaLevel: SixSigmaSigmaLevel(cell,lower_limit,upper_limit, shift)** A Six Sigma index, SixSigmaSigmaLevel calculates the Process Sigma Level with a shift. This statistic is defined as: $-\delta(\delta^{-1}(\frac{LowerSpecificationLimit\ -\hat{\mu}}{\hat{\sigma}} - Shift) +$ $1 - \delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift))$ where $\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process $\delta$is the standard normal cumulative distribution function, and δ^{-1} is the standard normal inverse cumulative distribution function.

		SigmaUpperBound: SixSigmaUpperBound(cell,number_stdev) A Six Sigma index, SixSigmaUpperBound calculates the Upper Bound as a specific number of standard deviations above the mean and is defined as: $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process. **SigmaYield: SixSigmaYield(cell,lower_limit,upper_limit,shift)** A Six Sigma index, SixSigmaYield calculates the Six Sigma Yield with a shift, or the fraction of the process that is free of defects. This statistic is defined as: $\delta^{-1}(\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift) -$ $\delta^{-1}(\frac{LowerSpecificationLimit-\hat{\mu}}{\hat{\sigma}} - Shift)$ where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function. **SigmaZLower: SixSigmaZLower(cell,lower_limit)** A Six Sigma index, SixSigmaZLower calculates the number of standard deviations of the process that the lower limit is below the mean of the process. This statistic is defined as: $\frac{\hat{\mu}-LowerSpecificationLimit}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process. **SigmaZMin: SixSigmaZMin(cell,lower_limit,upper_limit)** A Six Sigma index, SixSigmaZMin calculates the minimum of SigmaZLower and SigmaZUpper. This statistic is defined as: $\frac{MIN(\hat{\mu}-LowerSpecificationLimit,UpperSpecificationLimit-\hat{\mu})}{\hat{\sigma}}$ where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process. **SigmaZUpper: SixSigmaZUpper(cell,upper_limit)** SixSigmaZUpper(cell,upper_limit,simulation) A Six Sigma index, SigmaZUpper calculates the number of standard deviations of the process that the upper limit is above the mean of the process. This statistic is defined as: $\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}}$ Where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

Evaluations: Text Mining TFIDF Evaluations

Evaluation	Action	Definition
`detailedVocabulary`	Fit	Returns the number of times the terms, included in the Final List of Terms, appears in the document collection and the number of documents that include each term.
`docInfo`	Fit	Returns the # Characters and the # Terms per document.
`termCountInfo`	Fit	Returns the number of total terms, the % Reduction in Terms, the Final number of terms after preprocessing or reduction, and the setting for "maxVocabulary".
`vocabulary`	Fit	Returns the final list of terms. The number of terms included in this list is determined by the "maxVocabulary" parameter setting.
`zipfPlot`	Fit	Returns information to construct the Zipf Plot. This plots the number of times a term appears in the document collection.

Evaluations: Text Mining – Latent Semantic Analysis

Evaluation	Action	Definition
`conceptImportance`	Fit	Returns the Concept Importance table which lists each concept, its singular value, the cumulative singular value and the % singular value explained. (The number of concepts extracted is the minimum of the number of documents and the number of terms.) These values are used to determine which concepts should be used in the Concept – Document Matrix, Concept – Term Matrix and the Scree Plot according to the Users selection on the Representation tab.
`screePlot`	Fit	Returns information for constructing a Scree Plot
`termconceptMatrix`	Transform	Lists the most important concepts along the top of the matrix and most frequently appearing terms down the side of the matrix.
`termImportance`	Transform	Returns the most frequently appearing terms along with their Importance factor.
`vocabulary`	Fit	Returns terms contained in the input Term-Concept Matrix

Evaluations: Time Series – ARIMA and Lag Analysis

Evaluation	Action	Definition
`acfPlot`	Forecast	Lag Analysis: Plots the ACF (Autocorrelation).

`acvfPlot`	Forecast	Lag Analysis: Plots the ACVF (Autocovariance).
`autocorrelation`	Forecast	Lag Analysis: Returns the autocorrelation function
`autoCovariance`	Forecast	Lag Analysis: Returns the autocovariance function
`coefficientsInfo`	Fit	ARIMA: Returns the coefficients of the ARMA model.
`difference`	Forecast	Lag Analysis: Returns the differenced data.
`ljungBoxInfo`	Fit	ARIMA: Returns the Ljung-Box Test for Residuals information.
`loglikelihood`	Fit	ARIMA: Returns the LogLiklihood
`pacfPlot`	Forecast	Lag Analysis: Plots the PACF.
`partialAutocorrelation`	Forecast	Lag Analysis: Returns the partial autocorrelation function
`numIterations`	Fit	ARIMA: Returns the number of iterations completed.
`Residuals`	Transform	ARIMA: Returns the difference between the actual and predicted values.retur
`transformation`	Transform	ARIMA: Returns the predicted values using the fitted model when applied to a dataset.
`tsPlot`	Transform	Returns the time series plot: Actual vs Forecast.
`varCovarMatrix`	Fit	ARIMA: Returns the variance-covariance matrix.

Evaluations: Transformation Common Evaluations

Evaluation	Action	Definition
`transformation`	Transform	Returns the transformed dataset.

Evaluations: Transformation -- Summarization

Evaluation	Action	Definition
`histogram`	Transform	Returns information of summarization as a histogram.
`summary`	Transform	Returns summary information for Big Data.

Evaluations: Transformation -- Imputer

Evaluation	Action	Definition
`recordsWithMissingValues`	Transform	Returns records with missing values
`recordsToDelete`	Transform	Returns records that were deleted.

Evaluations: Transformation – Principal Components

Evaluation	Action	Definition
`principalComponents`	Transform	Returns the principal components
`principalEigenvalues`	Transform	Returns the eigenvalues.
`principalVariance`	Transform	Returns the Explained Variance percentage.
`tSquaredStatistic`	Transform	Returns Hotelling's t-squared statistic per record.
`qStatistic`	Transform	Returns the QStatistic per record.

Evaluations: Transformation -- Rescaler

Evaluation	Action	Definition
`statistics`	Fit	Returns the fitted statistics (shift and scale) for each rescaled variable.
`transformation`	Transform	Returns the transformed dataset.

Fitted Model ("fittedModel")

Used (only) when scoring a model. This section is similar to "datasets" but rather than refining imported data, this section defines a model that you can bind to when performing an "action" such as "forecast", "predict", "fit" or "transform".

In the example below, a previously fit linear regression model saved in PMML format (regression-linear-model.xml) is imported into RASON as "pmmlModelSrc", then bound to the model, "mlrModel", which is used to score the Boston Housing dataset (which was imported into RASON as "dataSrc" and then bound to "myData").

```
{
  comment: 'regression: linear model scoring from pmml',
  datasources: {
    dataSrc: {
      type: 'csv',
      connection: 'BostonHousingReg.txt'
    },
    pmmlModelSrc: {
      type: 'xml',
      content: 'pmml-model',
      connection: 'PMML\\regression-linear-model.xml'
    }
  },
  datasets: {
    myData: {
      binding: 'dataSrc'
    }
  },
  fittedModel: {
    mlrModel: {
      binding: 'pmmlModelSrc'
    }
  },
  actions: {
    myDataPrediction: {
      data: 'myData',
      fittedModel: 'mlrModel',
      action: 'predict',
      evaluations: [
        'prediction'
      ]
    }
  }
}
```

This section includes only one property, "binding".

Model Property	Example	Explanation
binding	"binding":"mySrc"	Binds to a previously fit model saved in either PMML or JSON format. (String property)

PreProcessor ("preProcessor")

This optional section may be used for preliminarily data preparation or to compute values of some properties, which are passed later, at parse-time, to the RASON DM engine. This section is parsed once, before the model is parsed.

In the example below, "numLeafRecords, is defined within the "preprocessor" section and is then referenced within the estimator, "treeEstimator", to set the parameter, "minNumRecordsInLeaves".

```
 preProcessor: {
    numLeafRecords: {
      formula: 'INT(MAX(1, ROWS(myTrainData) / 10))'
    }
  },
  estimator: {
    treeEstimator: {
      type: 'classification',
      algorithm: 'decisionTree',
      parameters: {
        priorProbMethod: 'EMPIRICAL',
        minNumRecordsInLeaves: 'numLeafRecords',
        maxNumNodes: 5,
        maxNumLevels: 3,
        maxNumSplits: 10,
        categoricalFeaturesNames: [ 'X1' ],
        prunedTreeType: 'MIN_ERROR'
      }
    }
  },
```

See the table below for the properties available in the formula section of your RASON model.

Data Property	Type	Explanation
name	"name": "parts"	Use this property to define the array name.
dimensions	"dimensions": [3,1] "dimensions": [3]	Defines a 1 – dimensional vertical array with 3 elements.

	"dimensions": [1,3] "dimensions": [3,2]	Defines a 1-dimensional vertical array with 3 elements. Defines a 2 – dimensional horizontal array with 3 elements. Defines a 2 – dimensional array with 3 rows and 2 columns. All arrays are 1 – based. If missing, array shape will be defined by the shape of the value property; however, for easier readability of the code, the use of the *dimensions* property is recommended.
value	"value": [1, 1, 1]	Sets the value of the array. While it is unlikely that this property would be required within formulas, as typically the value of an object will be computed by formula, it is permissible. See the example model, RGSpace2.json for an example. If dimensions property is missing, the shape of the variable array will be determined by the shape of the value property. However, it is recommended that the *dimensions* property be used for readability purposes.
formula	"formula": "5 + 2.5*temp2" "formula": "MATOP(Supply, 'min', '+', transpose(Demand))"	Enter a formula to calculate a result or array which will be used in a constraint, uncertain function or in the objective function.
comment	"comment": "partsReq array holds the number of parts required to produce each product"	Enter a comment here to describe the data.

JSON/XML Formats for DataFrames

As mentioned earlier, a DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). RASON DM extends the definition of a DataFrame to mark one or more columns as index.

XLMinerSDK: XLMiner::*DataFrame*; **SolverSDK**: SolverPlatform::Data::*CMemoryDataFrame*

We'll use the following example of DataFrame:

```
MyDF (2 x 3)

  integer     double     wstring

  IntCol (i)  DoubleCol  StringCol

Row1     1             1.5        A

Row2     2             2.5        B
```

This is DataFrame "MyDF" containing 2 rows and 3 columns. Column types are integer, double and wide string. Column names are "IntCol", "DoubleCol", "StringCol." "IntCol" is an index column. Row names are "Row1", "Row2."

Note: Currently, Row Names are directly available only in XLMinerSDK and not in SolverSDK. It can be considered as an explicit index column for most commonly used case of 2D tables, where 2 dimensions are uniquely defined by Row Names *and Col Names (or their ordinal indices). The purpose is to allow quick get-set operations such as DF[<row_name>,<col_name>]*

JSON

The most **complete format** for a JSON DataFrame with all optional and required fields explicitly provided may be found below.

```
{
 "name": "MyDF",
 "colNames": ["IntCol","DoubleCol","StringCol"],
 "rowNames": ["Row1","Row2"],
 "colTypes": ["integer","double","wstring"],
 "indexCols": ["IntCol"],
 "order": "col",
 "data": [[1,2],[1.5,2.5],["A","B"]]
}
```

The most **minimal format** for JSON DataFrame with all optional fields omitted and defaulted may be found below:

```
[[1,2],[1.5,2.5],["A","B"]]
```

The same data can be equivalently stored in a row-wise order:

```
{
 …
 "order": "row",
 "data": [[1,1.5,"A"],[2,2.5,"B"]]
}
```

The table below lists the available fields for a JSON DataFrame.

Field	Optional?	**Type**	**Possible Values**	Default
`name`	Yes	String		"DataFrame"
`colNames`	Yes	Array		["Col1",…,"ColN"]
`rowNames`	Yes	Array		["Row1",…,"RowN"]
`colTypes`	Yes	Array	{"integer","double","wstring"} for each column	"wstring" for each column
`indexCols`	Yes	Array	Valid column names	[] or null
`order`	Yes	String	{"row","col"}	"col"
`data`	No	2D Array		

Note: Currently, if the "colTypes" field is omitted, we assume string columns.

XML

The most **complete format** for an XML DataFrame with all optional and required fields explicitly provided:

Column-wise order:

```
<?xml version="1.0" encoding="utf-8"?>
<DataFrame name="MyDF" order="col">
    <Rows>
        <Row name="Row1"/>
        <Row name="Row2"/>
    </Rows>
    <Columns>
        <Column index="true" name="IntCol" type="integer">
            <Value>1</Value>
            <Value>2</Value>
        </Column>
        <Column index="false" name="DoubleCol" type="double">
            <Value>1.5</Value>
            <Value>2.5</Value>
        </Column>
        <Column index="false" name="StringCol" type="wstring">
            <Value>A</Value>
            <Value>B</Value>
        </Column>
    </Columns>
</DataFrame>
```

Row-wise order:

```
<?xml version="1.0" encoding="utf-8"?>
<DataFrame name="MyDF" order="row">
    <Rows>
        <Row name="Row1">
            <Value>1</Value>
            <Value>1.5</Value>
            <Value>A</Value>
        </Row>
        <Row name="Row2">
            <Value>2</Value>
            <Value>2.5</Value>
            <Value>B</Value>
        </Row>
    </Rows>
    <Columns>
        <Column index="true" name="IntCol" type="integer"/>
        <Column index="false" name="DoubleCol" type="double"/>
        <Column index="false" name="StringCol" type="wstring"/>
    </Columns>
</DataFrame>
```

The most **minimal formal** for JSON DataFrame with all optional fields omitted and defaulted:

Column-wise order:

```
<?xml version="1.0" encoding="utf-8"?>
<DataFrame>
```

```
    <Columns>
        <Column>
            <Value>1</Value>
            <Value>2</Value>
        </Column>
        <Column>
            <Value>1.5</Value>
            <Value>2.5</Value>
        </Column>
        <Column>
            <Value>A</Value>
            <Value>B</Value>
        </Column>
    </Columns>
</DataFrame>
```

Row-wise order:

```
<?xml version="1.0" encoding="utf-8"?>
<DataFrame order="row">
    <Rows>
        <Row>
            <Value>1</Value>
            <Value>1.5</Value>
            <Value>A</Value>
        </Row>
        <Row>
            <Value>2</Value>
            <Value>2.5</Value>
            <Value>B</Value>
        </Row>
    </Rows>
</DataFrame>
```

XSD Schema Definition:

Note: It's possible to infer the "order" ("col"/"order") property directly from the XML given the structure of "Rows"/"Columns" elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="orderType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="col"/>
      <xs:enumeration value="row"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="colType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="integer"/>
      <xs:enumeration value="double"/>
      <xs:enumeration value="wstring"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="DataFrame">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="Rows">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="0" name="Row">
```

```
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="Value" type="xs:string"/>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="Columns">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="0" name="Column">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="Value" type="xs:string"/>
                  </xs:sequence>
                  <xs:attribute name="index" type="xs:boolean"/>
                  <xs:attribute name="name" type="xs:string"/>
                  <xs:attribute name="type" type="colType"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="order" type="orderType"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

RASON

Import

We use our regular syntax for RASON “datasources” with the {"type":"json"} and {"type":"xml"}:

```
{
  "datasources": {
    "jsonSrc": {
      "type": "json", // "xml"
      "connection": "df.json", // "df.xml"
      "selection": "object1.object2",
      "content": "json-model", // "pmml-model"
    }
  },
  "datasets": {
    "myData": {
      "binding": "jsonSrc"
    }
  }
}
```

Fields:

- **"selection" property is *optional* and serves to access nested JSON/XML objects – see the example below. By *default*, we assume non-nested JSON/XML structure.**

```
{
  "x": {
    "y": {…df…}
  }
}
```

- "content" property is *optional* and serves to distinguish different objects that JSON/XML files can hold. If omitted, "table" is assumed.

The data source with the {"type":"json"} can contain the following serialized objects:

- ○ "content": "table" [DEFAULT] – table/dataframe in JSON format
- ○ "content": "json-model"– fitted DM model in JSON format

The data source with the {"type":"xml"} can contain the following serialized objects:

- ○ "content": "table" [DEFAULT] – table/dataframe in XML format
- ○ "content": "pmml-model"– fitted DM model in JSON format

RASON DM has additional types "content", which are not supported in any domain of RASON other than DM:

- ○ "content": "corpus" – text mining, corpus of documents
- ○ "content": "time-series" – single-variable time series data
- ○ "content": "itemset" – association rules, transactions in the itemset format

Rason Decision Flow Components

Introduction

Users can easily create **multi-stage decision flows**, with business rules, optimization, simulation and machine learning models. Decision flow stages can execute the **full range of predictive and prescriptive analytics**: DMN-compatible decision tables, Excel calculations, SQL operations, Monte Carlo simulations, mathematical optimizations, or machine learning training or prediction steps. Multi-dimensional data is automatically passed between stages in a standard, general "indexed data frame" form that maintains a dimensional information across statistic and machine learning transformations. Users can choose to run only the **stages that need updating**, easily determine the outcome of each stage and obtain results in JSON or OData form, at each stage or at the final stage. Each stage can be **written inline or can invoke a reusable model** via its interface. Models can be written in **RASON or Excel** with Analytic Solver and may be used in multiple decision flows. Using **automatic scheduling**, users can schedule a decision flow to run at fixed intervals or specify how recently updated they want each stage to be. RASON will automatically determine when to run each stage.

Decision Flow Components

The following components may be used when formulating either an inline decision flow or a decision flow invoking reusable models.

flow/flowName	"flow":"optDMWorkflow" "flowName":"optDMWorkflow"	Assigns a name to the decision flow.

inputParameters	"inputParameters":{
	"Number_to_build": {
	"type": "array",
	"value": "optStage.
	Number_to_build.
	finalValue,
	"comment": "description"
	}
	}
		Section heading where the input results from the stage are passed to the decision flow stage.

type		The "type" property is used to inform RASON of the type of data being passed for the given input parameter. Supported properties are: number, string or text, Boolean, array or dataset (for data mining flows only).
value		Use this property to pass the input value.
comment		Use this component to provide a brief description of the input.
invokeModel	"invokeModel":"rescaling-reusable" "invokeModel":	Use this property to invoke a resuable model from within a decision flow.

	"Name=MyExcelConnection"	Use "Name=MyExcelConnection" when the model is saved on a OneDrive account where MyExcelConnection is a named data connection on the user's MyAccount page. For more information on Data Connections in RASON, see the Data Connections section in the Rason User Guide.
modelDescription	"modelDescription":"This is a description of the decision flow."	Use this component to provide a brief description of the decision flow.
modelType	"modelType":"optimization"	Use this component to specify the problem type of the invoked model in a decision flow stage. Model Type can be "optimization", "simulation", "datamining" or "calculation". For more information on this component, see More on the Model Type Property within the RASON User Guide.

outputResults	"outputResults": {
	"Number_to_build":{
	"evaluations": ["finalValue"],
	"type": "array",
	"comment": "This is a comment"
	}
	}
		Section heading where the output results from the stage are reported.

evaluations		Use this component within "outputResults" to pass the output results within the decision flow stage.
type		The "type" property is used to inform RASON of the type of data being passed for the given input parameter. Supported properties are: number, string or text, Boolean, array or dataset (for data mining flows only).
comment		Use this component to provide a brief description of the output.

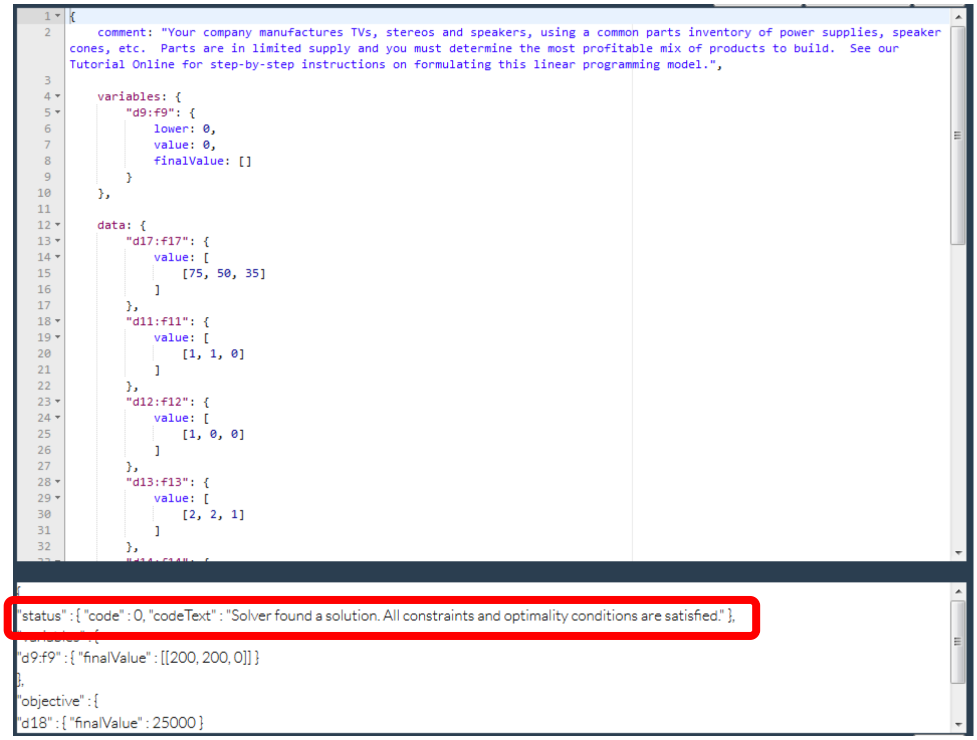
Solver Result Messages

Introduction

This chapter documents the Solver Result Messages that can be returned when you optimize a model or run a simulation, and discusses some of the characteristics and limitations of the Solver Engines.

Result Messages and Codes

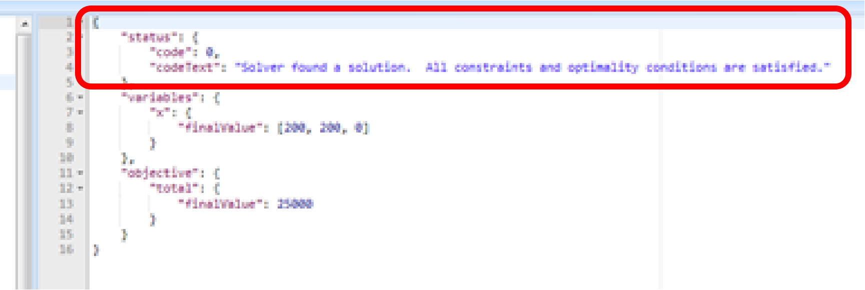
When the solution process completes, a Solver Result Message will be returned. If you are using the RASON.com Web ID, the result will appear at the bottom of the screen in the answer field next to "status".



```
{
    comment: "Your company manufactures TVs, stereos and speakers, using a common parts inventory of power supplies, speaker cones, etc.  Parts are in limited supply and you must determine the most profitable mix of products to build.  See our Tutorial Online for step-by-step instructions on formulating this linear programming model.",

    variables: {
        "d9:f9": {
            lower: 0,
            value: 0,
            finalValue: []
        }
    },

    data: {
        "d17:f17": {
            value: [
                [75, 50, 35]
            ]
        },
        "d11:f11": {
            value: [
                [1, 1, 0]
            ]
        },
        "d12:f12": {
            value: [
                [1, 0, 0]
            ]
        },
        "d13:f13": {
            value: [
                [2, 2, 1]
            ]
        },
        "d14:f14": {
```

```
{
"status" : { "code" : 0, "codeText" : "Solver found a solution. All constraints and optimality conditions are satisfied." },
"variables" : {
"d9:f9" : { "finalValue" : [[200, 200, 0]] }
},
"objective" : {
"d18" : { "finalValue" : 25000 }
```

If you are using the RASON desktop IDE, the answer will appear on the right, also net to "status".



```
{
    "status": {
        "code": 0,
        "codeText": "Solver found a solution.  All constraints and optimality conditions are satisfied."
    },
    "variables": {
        "x": {
            "finalValue": [200, 200, 0]
        }
    },
    "objective": {
        "total": {
            "finalValue": 25000
        }
    }
}
```

Values returned for "variables" and "objective" (and "constraints" if used) are the decision variable values and the objective function value for the *best* solution found.

Each Solver Result Message has a corresponding integer result code, (denoted as "code") as documented in this chapter. If you are using the REST API to call the RASON server, the endpoint `GET rason.net/api/model/id/result` (or QuickSolve endpoints `POST rason.net/api/optimize` or `POST rason.net/api/simulation`) will return this code and any final values specified by `finalValue: []`.

The RASON Server returns the integer result codes and displays the Solver Result Messages described in this section. The meanings of these messages have been generalized for the LP/Quadratic Solver, SOCP Barrier Solver, nonlinear GRG Solver, and Evolutionary Solver, and the conditions they may return. Please see the explanations of each message, especially for return code 0, “Solver found a solution.”

Plug-in Solver engines return the same codes and display the same messages as the built-in Solver engines whenever possible, but they can also return custom result codes (starting with 1000) and display custom messages, as described in their individual documentation.

Interval Global Solver

The Interval Global Solver is only available when an Excel file is loaded directly into the Solver SDK Platform using `prob.load` when called through a programming language or the desktop IDE. The Interval Global Solver is not available when called through the WEB IDE or the REST API. This engine can return three of these custom result codes and messages, described at the end of this section.

{"Exception": "You have 10757 variables. Your license allows 200 variables."}

This result is returned if you have not upgraded to the Pro or Platform tiers or if your license has expired. For further information about this exception, please call Frontline Systems at (775) 831-0300, or send an email to us at info@solver.com.

{"The remote server returned an error: (401) Unauthorized."}

This result is returned if your validation token is invalid. If, for any reason, you would like to be issued a new token, you can invalidate your existing authorization token by clicking the My Account link in the top right corner and then "invalidate". To obtain a new authorization token you must first log out. A new authorization token will be issued the next time you log in.

0. Solver found a solution. All constraints and optimality conditions are satisfied.

This means that the Solver has found the optimal or “best” solution under the circumstances. The exact meaning depends on whether you are solving a linear or quadratic, smooth nonlinear, global optimization, or integer programming problem, as outlined below. Solvers for non-smooth problems rarely if ever display this message, because they have no way of testing the solution for true optimality.

If you are solving a *linear* programming problem or a *convex quadratic* programming problem, the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. It is possible that there are other solutions with the *same* objective value, but all such solutions are linear combinations of the current decision variable values.

If you are solving a *linear* (LP), *convex quadratic* (QP) or *quadratically constrained* (QCP)*, or second order cone programming* (SOCP) problem, the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. It's possible that there are other solutions with the *same* objective value, but all such solutions are linear combinations of the current decision variable values.

If you are solving a *smooth nonlinear* optimization problem with no integer constraints, Solver has found a *locally* optimal solution: There is no other set of values for the decision variables *close to the current values* and satisfying the constraints that yields a better value for the objective. In general, there *may* be other sets of values for the variables, far away from the current values, which yield better values for the objective and still satisfy the constraints.

If you are using the Interval Global Solver for *global optimization* of a *smooth nonlinear* problem with no integer constraints, this means that the Solver has found the globally optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. But this is *subject to limitations* due to the finite precision of computer arithmetic that can, in rare cases, cause the Solver to "miss" a feasible solution with an even better objective value.

If you are solving a *mixed-integer* programming problem (any problem with integer constraints, this message means that the Branch & Bound method has found a solution satisfying the constraints (including the integer constraints) with the "best possible" objective value (but see the next paragraph). If the problem is linear or quadratic, the true integer optimal solution has been found. If the problem is smooth nonlinear, the Branch & Bound process has found the best of the locally optimal solutions found for subproblems by the nonlinear Solver.

1. Solver has converged to the current solution. All constraints are satisfied.

This means that Solver has found a series of "best" solutions that satisfy the constraints, and that have very similar objective function values; however, no single solution strictly satisfies the Solver's test for optimality. The exact meaning depends on whether you are solving a smooth nonlinear problem with the LSGRG Solver or the Interval Global Solver, or a non-smooth problem with the Evolutionary Solver.

When the LSGRG Solver or the Interval Global Solver is being used, this message means that the objective function value is changing very slowly as the Solver progresses from point to point. More precisely, the Solver stops if the absolute value of the relative (i.e. percentage) change in the objective function, in the last few iterations, is less than the Convergence tolerance on the Task Pane Engine tab. A *poorly scaled* model is more likely to trigger this stopping condition, even if scaling = True in `engineSettings`. If you are sure that your model is well scaled, you should consider why it is that the objective function is changing so slowly.

When the Evolutionary Solver is being used to solve a nonsmooth model, this message means that the "fitness" of members of the current population of candidate solutions is changing very slowly. More precisely, the Evolutionary Solver stops if 99% or more of the members of the population have "fitness" values whose relative (i.e. percentage) difference is less than the Convergence tolerance on the Task Pane Engine tab. The "fitness" values incorporate both the objective function and a penalty for infeasibility, but since the Solver has found some feasible solutions, this test is heavily weighted towards the objective function values. If you believe that the

Solver is stopping prematurely when this test is satisfied, you can make the Convergence tolerance smaller, but you may also want to increase the Mutation Rate and/or the Population Size, in order to increase the diversity of the population of trial solutions.

2. Solver cannot improve the current solution. All constraints are satisfied.

This means that the Solver has found solutions that satisfy the constraints, but it has been unable to further improve the objective, even though the tests for optimality ("Solver found a solution") and convergence ("Solver converged to the current solution") have not yet been satisfied. The exact meaning depends on whether you are solving a smooth nonlinear problem with the Standard LSGRG Solver, a global optimization problem with the Interval Global Solver, or a non-smooth problem with the Evolutionary Solver.

When the LSGRG Solver is being used, this message occurs very rarely. It means that the model is *degenerate* and the Solver is probably *cycling*. One possibility worth checking is that some of your constraints are redundant, and should be removed.

When the Interval Global Solver is being used, this message is more common. It means that the Solver has not found an "improved global solution" (a feasible solution with an objective value better than the currently best known solution), in the amount of time specified for the `maxTimeNoImprove: True` within `engineSettings`. The reported solution is the best one found so far, but the search space has not been fully explored. If you receive this message, and you are willing to spend more solution time to have a better chance of "proving" global optimality, increase the value of `maxTimeNoImprove`.

When the Evolutionary Solver is being used, this message is much more common. It means that the Solver has been unable to find a new, better member of the population whose "fitness" represents a relative (percentage) improvement over the current best member's fitness of more than the `intTolerance` option setting in `engineSettings`, in the amount of time specified by the `maxTimeNoImp` option also within `engineSettings`. Since the Evolutionary Solver has no way of testing for optimality, it will normally stop with either "Solver converged to the current solution" or "Solver cannot improve the current solution" if you let it run for long enough. If you believe that this message is appearing prematurely, you can either decrease the setting for `intTolerance` (even setting it to zero), or increase the value for `maxTimeNoImp` .

3. Stop chosen when the maximum iteration limit was reached.

This result is returned when the Solver has completed the maximum number of iterations, or trial solutions, set for `iterations` in `engineSettings` . The default setting for this option is *unlimited.*

If you are solving a *mixed-integer* programming problem (any problem with integer constraints), this message is relatively unlikely to appear. The Evolutionary Solver uses `maxSubproblems` and `maxIntegerSols` specified in `engineSettings`, and the Branch & Bound method (employed by the other Solver engines on problems with integer constraints) uses `maxSubproblems` and `maxIntegerSols` options also within `engineSettings`, to control the overall solution process. The count of iterations against which the Iteration limit is compared is reset on each new subproblem, so this limit usually is not reached.

If you are using Stochastic Decomposition to solve for a stochastic linear model, this status will be returned if the algorithm performs over 5,000 iterations.

4. The objective (Set Cell) values do not converge.

This result is returned when the Solver is able to increase (if you are trying to Maximize) or decrease (for Minimize) without limit the value calculated by the objective, while still satisfying the constraints. Remember that, if you've `set type: "Minimize"` within `objective`, the objective may take on negative values without limit unless this is prevented by the constraints or bounds on the variables. Set the `assumeNonneg: True` within `engineSettings` to impose >= 0 bounds on all variables.

If the objective is a linear function of the decision variables, it can *always* be increased or decreased without limit (picture it as a straight line), so the Solver will seek the extreme value that still satisfies the constraints. If the objective is a nonlinear function of the variables, it may have a "natural" maximum or minimum (for example, =A1*A1 has a minimum at zero), or no such limit (for example, =LOG(A1) increases without limit).

If you receive this message, you may have forgotten a constraint, or failed to anticipate values for the variables that allow the objective to increase or decrease without limit. The final values for the variable cells, the constraint left hand sides and the objective should provide a strong clue about what happened.

The Evolutionary Solver *never* displays this message, because it has no way of systematically increasing (or decreasing) the objective function, which may be non-smooth. If you have forgotten a constraint, the Evolutionary Solver *may* find solutions with very large (or small) values for the objective – thereby making you aware of the omission – but this is not guaranteed.

5. Solver could not find a feasible solution.

This result is returned when the Solver could not find *any* combination of values for the decision variables that allows all of the constraints to be satisfied *simultaneously*. If you are using the LP/Quadratic Solver or the SOCP Barrier Solver, and the model is well scaled, the Solver has determined for *certain* that there is no feasible solution.

If you are using the standard LSGRG Solver, the GRG method (which always starts from the initial values of the variables) was unable to find a feasible solution; but there could be a feasible solution far away from these initial values, which the Solver might find if you run it with different initial values for the variables. To set initial values for the variables, use the value property. In the example below the initial values of all three variables contained in the `x` array are being set to "1" via the `value` property.

```
variables: {
        x: {
            dimensions: [3],
            value: 0,
            lower: 0,
            finalValue: []
        }
```

Alternatively, the initial values of each variable could also have been set using

`value: [0, 0, 0].`

If you are using the Interval Global Solver, this message means that the Solver could find no feasible solutions after a systematic exploration of the search space. The Interval Global Solver is designed to "prove feasibility" as well as global optimality, and there is very likely *no* feasible solution; but this is *subject to limitations* due to the finite precision of computer arithmetic that can, in rare cases, cause the Solver to "miss" a solution.

If you are using the Evolutionary Solver, the evolutionary algorithm was unable to find a feasible solution; it might succeed in finding one if you run it with different initial values for the variables (see above for an example) and/or increase the setting for `precision` within `engineSettings` (which reduces the infeasibility penalty, thereby allowing the evolutionary algorithm to explore more "nearly feasible" points).

If you are solving a problem with chance constraints using simulation optimization, this message means that the Solver could find no solution that satisfies these constraints to the chance measures (such as 95%) that you specified. If you 'relax' the chance measures (to say 90%) and solve again, it's possible that a feasible solution will be found. For robust optimization, see result codes 26 through 29.

In any case, you should first look for conflicting constraints, i.e. conditions that *cannot* be satisfied simultaneously. Most often this is due to choosing the wrong relation (e.g. <= instead of >=) on an otherwise appropriate constraint.

6. Solver stopped at user's request.

This result is returned only when the REST API endpoint, *POST RASON.net/api/model/id/stop* is called.

7. The linearity conditions required by this Solver engine are not satisfied.

This result is returned if you've selected the LP/Quadratic Solver and the Solver's tests determine that the constraints are not linear functions of the variables or the objective is not a linear or convex quadratic function of the variables; *or* if you've selected the SOCP Barrier Solver and the Solver's tests determine that the constraints or the objective are not linear or convex quadratic functions of the variables.

If you receive this message, examine the formulas for the objective and constraints for nonlinear or non-smooth functions or operators applied to the decision variables or set `transformNonsmooth: True` within `modelSettings` to have Solver attempt to transform your nonlinear or nonsmooth model into a linear model. For more information on Nonsmooth Model Transformation see this option in the `modelSettings` explanation above.

8. {"Exception": "You have _____ variables. Your license allows _____ variables."}

This result is returned when the Solver determines that your model is too large for the selected Solver engine within your Account tier. You'll have to select – or possibly upgrade to – another Solver engine appropriate for your problem, or else reduce the number of variables, constraints, or integer variables to proceed.

9. Solver encountered an error value in a target or constraint cell.

This message appears when the Solver SDK Platform (on the RASON server or on your desktop) evaluates the formulas in your RASON model and discovers an error value while calculating the objective function, uncertain function or one of your constraints. Most often, a more specific message will appear instead of "Solver encountered an error value in a (nonspecific) target or constraint cell." At a minimum, the message will say "Error value returned at *line number*," where *line number* tells you exactly where the error was encountered. Other messages may tell you more about the error. The general form of the message is:

***Error condition* at *line number.* Edit your formulas.** *Error condition* is one of the following:

Floating point overflow	Invalid token
Runtime stack overflow	Decision variable with formula
Runtime stack empty	Decision variable defined more than once
String overflow	Missing Diagnostic/Memory evaluation
Division by zero	Unknown function

Unfeasible argument	Unsupported Excel function
Type mismatch	Excel error value returned
Invalid operation	Non-smooth special function

See also result code 21, “Solver encountered an error computing derivatives,” and result code 12, with messages that can appear when the Interpreter first analyzes the formulas in your model.

“Floating point overflow” indicates that the computed value is too large to represent with computer arithmetic; “String overflow” indicates that a string is too long to be stored in a cell. “Division by zero” would yield #DIV/0! on the worksheet, and “Unfeasible argument” means that an argument is outside the domain of a function, such as =SQRT(A1) where A1 is negative.

“Unknown function” appears for functions whose names are not recognized by the Interpreter, such as user-written functions. “Unsupported Excel function” appears for the few functions that the Interpreter recognizes but does not support.

The Evolutionary Solver rarely, if ever, display this message – since they maintain a *population* of candidate solutions and can generate more candidates without relying on derivatives, they can simply discard trial solutions that result in error values in the objective or the constraints. If you have a model that frequently yields error values for trial solutions generated by the Solver, and you are unable to correct or avoid these error values by altering your formulas or by imposing additional constraints, you can still use the Evolutionary Solver to find (or make progress towards) a “good” solution.

10. Stop chosen when the maximum time limit was reached.

This result is returned when Solver has run for the maximum time (number of seconds) specified for `maxTime` within `engineSettings`. The default setting for this option is *unlimited.*

11. There is not enough memory available to solve the problem.

This message appears when the Solver could not allocate the memory it needs to solve the problem. If you are calling the Solver SDK through the RASON modeling language to solve your model, you are likely to notice that solution times have greatly slowed down, and the hard disk activity light in your PC has started to flicker before you see this result.

You can save some memory by closing any Windows applications other than the programming language you are using or the Desktop IDE and closing programs that run in the System Tray

No model inputs defined.

This message means that the internal “model” (information about the variable cells, objective, constraints, Solver options, etc.) is not in a valid form. An “empty” or incomplete Solver model, perhaps one with no objective and no constraints other than bounds on the variables, can cause this message to appear.

14. Solver found an integer solution within tolerance. All constraints are satisfied.

If you are solving a mixed-integer programming problem (any problem with integer constraints) with a *non-zero value* for the `intTolerance` within `engineSettings`, the Branch & Bound method has found a solution satisfying the constraints (including the integer constraints) where the relative difference of this solution’s objective value from the *true* optimal objective value does not exceed the integer Tolerance setting. (For more information, see the explanation for Integer Tolerance in the Engine Options section within the chapter, "Rason Model

Components"). This may actually *be* the true integer optimal solution; however, the Branch & Bound method did not take the extra time to search all possible remaining subproblems to “prove optimality” for this solution. If all subproblems *were* explored (which can happen even with a non-zero `intTolerance` setting in some cases), the result "Solver found a solution. All constraints are satisfied" (result code 0, shown earlier in this section) will be returned.

15. Stop chosen when the maximum number of feasible [integer] solutions was reached.

If you are using the Evolutionary Solver, this result is returned when the Solver has found the maximum number of feasible solutions (values for the variables that satisfy all constraints) allowed by the `maxFeasibleSols` option setting within `engineSettings`. You may increase the value for `maxFeasibleSols` or leave this setting at the default, *unlimited.*

If you are using one of the other Solver engines on a problem with integer constraints, this result is returned when the Solver has found the maximum number of integer solutions (values for the variables that satisfy all constraints, including the integer constraints) allowed by the `maxIntegerSols` option setting within `engineSettings`. You may increase the value for `maxIntegerSols` or leave the option setting at the default, *unlimited*. But you should also consider whether the problem is formulated correctly, and whether you can add constraints to "tighten" the formulation.

16. Stop chosen when the max number of feasible [integer] subproblems was reached.

If you are using the Evolutionary Solver, this result is returned when the Solver has explored the maximum number of subproblems specified for `maxSubproblems` within `engineSettings`. You may increase the value for the `maxSubproblems`, leave the option setting at its default, *unlimited.*

If you are using one of the other Solver engines on a problem with integer constraints, this result is returned when the Solver has explored the maximum number of integer subproblems (each one is a "regular" Solver problem with additional bounds on the variables) specified for `maxSubproblems` within `engineSettings`. You may increase the value for `maxSubproblems` or leave the options setting at its default, *unlimited.* But you should also consider whether the problem is formulated correctly, and whether you can add constraints to "tighten" the formulation.

17. Solver converged in probability to a global solution.

If you are using the multistart methods for global optimization, with the standard LSGRG solver, or a field-installable nonlinear Solver engine, this result is returned when the multistart method's Bayesian test has determined that all of the locally optimal solutions have *probably* been found; the solution displayed on the worksheet is the best of these locally optimal solutions, and is *probably* the globally optimal solution to the problem.

The Bayesian test initially assumes that the number of locally optimal solutions to be found is equally likely to be 1, 2, 3, … etc. up to infinity, and that the relative sizes of the regions containing each locally optimal solution follow a uniform distribution. After each run of the standard LSGRG Solver or field-installable Solver engine, an updated estimate of the most probable total number of locally optimal solutions is computed, based on the number of subproblems solved and the number of locally optimal solutions found so far. When the number of locally optimal solutions actually found so far is within one unit of the most probable total number of locally optimal solutions, the multistart method stops and displays this message.

18. All variables must have both upper and lower bounds.

If you are using the Interval Global Solver, this message is returned if you have not defined lower and upper bounds on all of the decision variables in the problem. You *must* define bounds on all variables in order to use this engine. You should add the missing bounds using the `lower` and `upper` properties within your variable array definition, and try again.

If you are using the Evolutionary Solver or the multistart methods for global optimization, and you have not set `requireBounds: False` within `engineSettings` (it is set to True by default), this message will also appear. You should add the missing bounds using the `lower` and `upper` properties within your variable array definition, and try again.

In the example code below, lower bounds of 1 and upper bounds of 10 are applied to each of the three elements in the x array.

```
variables: {
        x: {
            dimensions: [3],
            value: 1,
            lower: 10,
            finalValue: []
        }
```

Alternatively, unique lower and upper bounds for each variable could be specified using:

```
lower: [1, 2, 3],
upper:  [10, 11, 12],
```

Lower bounds of zero can be applied to all unbounded variables by using the `assumeNonneg` option within `engineSettings`. For the Evolutionary Solver or the multistart methods, such bounds are not absolutely required (you can set `requireBounds:   False`), but they are a practical necessity if you want the Solver to find good solutions in a reasonable amount of time.

19. Variable bounds conflict in binary or alldifferent constraint.

This result is returned if you have both a binary or alldifferent constraint on a decision variable and a <= or >= constraint on the same variable (that is inconsistent with the binary or alldifferent specification), or if two or more of the same decision variables appear in more than one alldifferent constraint. Binary integer variables always have a lower bound of 0 and an upper bound of 1; variables in an alldifferent group always have a lower bound of 1 and an upper bound of N, where N is the number of variables in the group. You should check that the binary or alldifferent constraint is correct, and ensure that alldifferent constraints apply to non-overlapping groups of variables. If a <= or >= constraint causes the conflict, remove it if possible and try to solve again.

20. Lower and upper bounds on variables allow no feasible solution.

This result is returned if you've defined lower and upper bounds on a decision variable, where the lower bound is greater than the upper bound. This (obviously) means there can be no feasible solution, but most Solver engines will detect this condition before even starting the solution process, and display this message instead of "Solver could not find a feasible solution" to help you more quickly identify the source of the problem.

21. Solver encountered an error computing derivatives.

This message appears when the Interpreter in Solver SDK Platform encounters an error when computing derivatives via automatic differentiation. The most common cause of this message is a non-smooth function in your objective, uncertain function or constraints, for which the derivative is undefined. But in general, automatic differentiation is somewhat more strict than finite differencing: As a simple example, =SQRT(test) evaluated at test=0 will yield this error message when the Solver is using automatic differentiation (since the derivative of the SQRT function is algebraically undefined at zero).

22. Variable appears in more than one cone constraint.

This result is returned if the same decision variable appears in more than one cone constraint. You can define as many cone constraints as you want, but each one must contain a different group of decision variables.

23. Formula depends on uncertainties, must be summarized or transformed. Learn more using the Solver Model dialog Diagnosis tab.

This result is returned if you've defined constraints or an objective computed by formulas that depend on uncertain parameters. Each such formula represents an *array* of sample values, one for each realization of the uncertainties. For your model to be well-defined, the objective or constraint must either be *summarized* to a single value (such as a mean or percentile value) or *transformed* into a set of single-valued constraints (through an automatic transformation in the Solver Model dialog).

To correct the problem, you can (i) use the `chanceType` property to define the constraint as a *chance* constraint or the objective as an *expected value* or *risk measure* objective (for more information see the *Constraints* and *Objective* sections within the **Rason Model Components** chapter), or (ii) edit the formula so that the objective formula value is computed by a PSI Statistics function such as PsiMean() or PsiPercentile().

25. Simulation optimization doesn't handle models with recourse decisions.

This result is returned if you've defined a recourse decision variable, but you've set `simulationOptimization:  True` within `modelSettings`. Simulation optimization, as defined in the academic literature and as implemented by Frontline Systems doesn't support the concept of recourse decision variables. To solve a problem with recourse decisions, you'll need to solve using stochastic programming and robust optimization methods, both of which *do* support recourse decision variables.

26. Solver could not find a feasible solution to the robust chance constrained problem.

This result is returned when you solve a model with uncertainty and chance constraints using robust optimization. When you do this, the Solver transforms your original model with uncertainty into a *robust counterpart* model that is a conventional optimization problem without uncertainty.

This message means that the Solver could not find a feasible solution to the robust counterpart problem. It does not *necessarily* mean that there is no feasible solution to the original problem; the robust counterpart is an *approximation* to the problem defined by your chance constraints that may yield conservative solutions which *over-satisfy* the chance constraints.

When this result is returned, try setting `chanceAutoAdjust: True` within `modelSettings` and resolving. The Solver will then re-solve the problem, automatically adjusting the sizes of robust optimization *uncertainty sets* created for the chance constraints, in an effort to find a feasible solution.

If the same result is returned, you should proceed as described for result code 5, “Solver could not find a feasible solution:” Look for conflicting constraints, i.e. conditions that *cannot* be satisfied simultaneously, perhaps due to choosing the wrong relation (e.g. <= instead of >=) on an otherwise appropriate constraint.

27. Solver found a conservative solution to the robust chance constrained problem. All constraints are satisfied.

This result may be returned when you solve a model with uncertainty and chance constraints using robust optimization. When you do this, the Solver transforms your original model with uncertainty into a *robust counterpart* model that is a conventional optimization model without uncertainty.

The message means that the Solver found an optimal solution to the robust counterpart model, but when this solution was tested against your *original* model (using Monte Carlo simulation to test satisfaction of the chance constraints), the solution *over-satisfied* the chance constraints; this normally means that the solution is 'conservative' and the objective function value can be further improved.

When this result is returned, try setting `chanceAutoAdjust: True` within `modelSettings` and resolving. The Solver will then re-solve the problem, automatically adjusting the sizes of robust optimization *uncertainty sets* created for the chance constraints, in an effort to find a feasible solution.

An alternative course of action is to *manually* adjust the Chance measures of selected chance constraints, and re-solve the problem. The automatic improvement algorithm uses general-purpose methods to find an improved solution; you may be able to do better by adjusting Chance measures based on your knowledge of the problem.

28. Solver has converged to the current solution of the robust chance constrained problem. All constraints are satisfied.

This result may be returned when you solve a model with uncertainty and chance constraints using robust optimization, and you've set `chanceAutoAdjust: True` within `modelSettings`. It means that the Solver has found the best 'improved solution' it can; the normal constraints are satisfied, and the chance constraints are satisfied to the Chance level that you specified in the Solver Parameters dialog.

This is usually a very good solution, but it does not rule out the possibility that you may be able to find an even better solution by manually adjusting Chance measures based on your knowledge of the problem, and re-solving.

999. Unexpected error. Please contact Technical Support.

This status signifies that an unexpected exception has occurred within Solver. If this status is returned, please contact our technical support team at support@solver.com.

Interval Global Solver Result Messages

The Interval Global Solver can return many of the standard result codes and Solver Result Messages described above, but it can also return one of three custom result codes and messages, as described below.

1000. Interval Solver requires strictly smooth functions.

The Interval Global Solver considers the 'special' functions ABS, IF, MAX, MIN or SIGN nonsmooth. If this message is returned, either reformulate model so that these functions are not used or select a different engine.

1001. Function cannot be evaluated for given real or interval arguments.

This message may appear (instead of "Solver encountered an error value…") if the Interval Global Solver encounters an arithmetic operation or function that it cannot evaluate for the current values of the decision variables. Recall that the Interval Global Solver evaluates formulas over intervals such as [1, 2] as well as real numbers. In the course of seeking a solution, the Solver may have to evaluate a formula that (for example) involves division by an *interval containing zero*, or the square root of an *interval containing negative values*, which yield errors. If you receive this message, try adding constraints, or adjusting the right hand sides of existing constraints to eliminate the problem.

For example, if you have trouble with a constraint (within your Excel model) such as A1 >= 0, try a constraint such as A1 >= 0.0001 instead.

1002. Solution found, but not proven globally optimal.

This message indicates that the Interval Global Solver has systematically explored the solution space and has found a solution that is very probably the global optimum, but it has not been able to "prove global optimality." Most often, this means that there is more than a tiny difference between this solution's objective value and the *best bound* on the global optimum’s objective value that the Solver has been able to find.

RASON DMN/FEEL at Conformance Level 3

Introduction

The latest version of RASON Decision Services supports DMN at Conformance Level 3 (CL3).

Creating independent DMN/Feel models

In the latest version of RASON Decision Services, DMN/Feel functionality no longer requires Excel formulas when representing a DMN decision model. The entire model may now be represented using only FEEL formulas, which are referred to as **literal expressions**. Such models are entirely independent of Excel syntax. These models are referred to as **pure DMN models**. Notice that pure DMN models can only be decision/calculation models. Currently, Rason Decision Services does not support optimization, simulation or data mining models as pure DMN models.

The main consequence of avoiding Excel formulas is to preserve the authentic DMN/Feel types in formula assignments.

For example,

dt: { feelFormula: "date('05-05-2021')" }

preserves the specific Feel type 'feel date' in the variable **dt**, so we may use in a later feelFormula: "**dt**.day".

Download DMN examples from the Editor tab at www.RASON.com by clicking the "Download Example Data" icon on the ribbon.

List data and related functions

A **list of elements** is a data structure that holds multiple elements.

For example:

D1: { value: [1, 5, 2] } is a list of 3 *scalar* elements, which is exactly the same as a 1D array in RASON Decision Services.

D2: { value: [[1,5], [7,3], [9,4]] } is a list of 3 *list* elements. Every element of the list D2 is another list of length 2. Because all list elements of D2 are identical (the same length), the structure D2 is exactly the same as the 2D array in RASON Decision Services.

DMN requires that all elements of a list must be of the same type. A "custom" type offers more flexibility. For more information on Custom Types in RASON, see the chapter *Custom Type Definitions* in the RASON User Guide. Lists may be simple 1D or 2D arrays or have custom types attached to them.

The goal of this example is to add a new record to the existing list using the append function.

```
{
  "comment": "Example of a list with typeDef collection",
  "typeDefs": {
    "tLoan": {
      "language": 'feel',
      "components": ['principal', 'rate', 'termMonths'],
      "types": ['number', 'number', 'number']
    },
    "tLoanList: {
      "language": 'feel',
      "isCollection": true,
      "typeRef": 'tLoan'
    }
  },
  "data: {
    "loan": {
      "type": 'tLoan',
      "value": [300000, 0.02, 360],
      "binding": 'get'
    },
    "loanList": {
      "type": 'tLoanList',
      "value": [[600000, 0.0275, 360], [300000, 0.03, 360]]
    }
  },
  "formulas": {
    "result": {
      "feelFormula": "append(loanList, loan)",
      "finalValue": [] }
    }
  }
}
```

The collection of records **tLoanList** is a collection of records of the custom component type **tLoan**. The variable **loan** is of type **tLoan** and is used to represent a single record in the **list** variable. Though the value of the variable **loan** is initialized as [300000, 0.02, 360] and looks like a list it is *not* a list but a component structure/record of type **tLoan**. By default, this vector will be used in computations unless a new vector of values is passed as query parameters to the RASON model.

Since the **list** variable of type **tLoanList** is a collection of records of type **tLoan**, it represents a table with records. It's important to understand that each element of the list **loanList** is not another list, but an element of the custom type **tLoan**.

The full list is returned for **result: { finalValue: [] }**.

Because the custom type is preserved in the variable result, future formulas may utilize result[1], result.rate, or result[1].rate to extract specific information.

- result[1] will return the first record as tLoan.
- result.rate will return the entire rate column as a vertical 1D array without custom type.
- result[1].rate will return the rate of the first record as a scalar number.

See the listed example functions below implemented for DMN CL3 for both type and non-typed lists. These functions and restricted to 1D and 2D array structures.

count(list) returns integer	count([1,2,5]) = 3
max(list) returns list elem type	max([1,2,5]) = 5

max(num1, num2,…) returns number	max(1,2,5) = 5
min(list) returns list elem type	min([1,2,5]) = 1
min(num1, num2,…) returns number	min(1,2,5) = 1
sum(list) returns number	sum([1,2,5]) = 8
sum(num1, num2,…) returns number	sum(1,2,5) = 8
roundUp(n, scale) returns number	Returns n with given scale, rounded up.
	roundup(5.5, 0) = 6; roundUp(-5.5, 0) = -6
	roundUp(1.121, 2) = 1.13; roundUp(-1.126, 2) = -1.13.
roundDown(n, scale) returns number	Returns n with given scale, rounded down.
	rounddown(5.5, 0) = 5; rounddown(-5.5, 0) = -5
	rounddown(1.121, 2) = 1.12; rounddown(-1.126, 2) = -1.12.
roundHalfUp(n, scale) returns number	Returns n with given scale; rounded up.
	roundHalfUp(5.5, 0) = 6; round HalfUp(-5.5, 0) = -6
	roundHalfUp(1.121, 2) = 1.12; roundHalfUp(-1.126, 2) = -1.13
roundHalfDown(n, scale) returns number	Returns n with given scale; rounded down.
	roundHalfDown(5.5, 0) = 5; round HalfUp(-5.5, 0) = -5
	roundHalfUp(1.121, 2) = 1.12; roundHalfUp(-1.126, 2) = -1.13

Signatures for other statistics functions, such as mean, median, mode, product, stdev, are the same as sum, i.e. mean(num1, num2, …).

and(list), **all**(list) returns Boolean

and(bool1, bool2,…), **all**(bool1, bool2,…) returns Boolean

or(list), **any**(list) returns Boolean

or(bool1, bool2,…), **any**(bool1, bool2,…) returns Boolean

sublist(list, start pos, [length]) returns list	sublist([1,2,5], 2) = [2,5]
append(list, elem1, elem2,…) returns list	append([1,2,5], 0) = [1,2,5,0]
concatenate(list1, list2,…) returns list	concatenate([1,2,5], [3,4]) = [1,2,5,3,4]
insertBefore(list, pos, elem) returns list	insertBefore([1,2,5], 3, 0) = [1,2,0,5]
listContains(list, elem) returns Boolean	listContains([1,2,5], 0) = false
remove(list, pos) returns list	removes([1,2,5], 3) = [1,2]
reverse(list) returns list	reverse([1,2,5]) = [5,2,1]
indexOf(list, match) returns list	indexOf([1,2,1,2,3], 2) = [2,4]
union(list1, list2,…) returns list	union([1,2,3], [1,2,5]) = [1,2,3,5]
distinctValues(list) returns list	distinctValues([1,2,1,2,3]) = [1,2,3]
flatten(list) returns list	flatten([1,2], [2,3]) = [1,2,2,3]
listReplace(list, position/match, newItem) returns	listReplace ([2,4,7,8],3,6) = [2,4,6,8]

list, number or Boolean function(item, newItem),	listReplace([2,4,7,8,function(item, newItem item < newItem,5) = [5,5,7,8]
any element including null	

Formatting results of box objects as custom types

Decision tables are key box objects in decision modeling. A decision table may have one or more outputs.

- If the result is a single successful rule evaluation, depending on the single or many outputs, we currently return either a scalar value or a horizontal vector.
- If the result is multiple successful rule evaluations (with the Collect hit policy) and the output is single, we return a vertical vector.
- In case of many outputs and many successful rule evaluations, we return a 2D array which resembles a table with columns for each output and records for each success. In any case, the result is either scalar or pure array and we can reference this information in later formulas by the standard rules: scalars as scalars and arrays by whole names or by the index operator.

DMN CL3 introduces a way of formatting the results of decision tables and box functions through custom types. This approach allows users to reference the result more efficiently. Here is an example in which a decision table result is formatted.

```
{
  "typeDefs": {
    "tParkingFee": {
      "language": "FEEL",
      "components": ['parkingFee','durVal'],
      "types": ['number','duration']
    }
  },
  "decisionTables": {
    "tblParkingFee": {
      "inputs": ["dtDuration"],
      "outputs": ["durVal", "parkingFee"],
      "refTypes": ["duration", "duration", "number"],
      "rules": [
        ["<'PT20M'", "duration(dtDuration)", 0],
        ["['PT20M'..'PT1H')", "duration(dtDuration)",
            "2 *ceiling(duration(dtDuration)/duration('PT20M'))"],
        ["['PT1H'..'PT4H')", "duration(dtDuration)", "6
        *ceiling(duration(dtDuration)/duration('PT1H'))"],
        [">='PT4H'", "duration(dtDuration)",
        "30*ceiling(duration(dtDuration)/duration('P1D'))"]],
      "hitPolicy": "U",
      "resultType": "tParkingFee"
    }
  },
  "data": {
    "dur": { value: "PT25M", comment: "period" }
  },
  "formulas": {
    "fee": { feelFormula: "tblParkingFee(,,dur)", finalValue: [] },
    "res": { feelFormula: "fee.durVal.minutes", finalValue: [] }
  }
}
```

The decision table (**tblParkingFee**) has two outputs (**durVal** and **parkingFee**) and a hit policy of 'U'. The result is a single record with two columns. We define a custom component type **tParkingFee** with the same component names as the outputs of the decision table. Then we set the table property **resultType**: **"tParkingFee"** to that custom type. The results, **durVal** and **parkingFee** are formatted using **refTypes**, "duration" and "number", respectively.

This example uses only feelFormulas in order to preserve the Feel types. The first formula assigns the decision result to the variable "**fee**", which is a 1D array of 2 elements – the outputs. However, this time "fee" has the **tParkingFee** custom type attached to it. Without the custom type, the variable "**fee**" may be referenced only by its name or through the index operator.

With the more flexible custom type and reference, only the desired component in the next formula **fee.durVal** is required. Since feel formulas preserve Feel types, the component **fee.durVal** is of type **duration**. The number of minutes that the car was parked can be extracted from the duration: **feelFormula: "fee.durVal.minutes".**

See the chapter *Defining Decision Tables in RASON* within the RASON User Guide for more information on Decision Tables.

Appendix: List of Examples

DMN Feel Date Time. json

DMN List example. json

DMN List with typeDef collection. json

DMN DT with typeDef result. json

DMN Box Fun with typeDef result. json

DMN Context example. Json

Appendix

Microsoft LET Function

The newly introduced Micorsoft Excel LET is supported in RASON Decision Services models.

In Excel, the LET function assigns a name to a calculation result which allows for the storage of intermediary calculations, values or defined names within a formula. See Microsoft's Office Support for a complete documentation of this function.

LET Function

To use this function, pairs of names and associated values are defined, up to 126 pairs. The last argument is a calculation that uses all defined names, for example: =LET(x, 2, x + 9) where LET evaluates to 11. In the example below, this function is used in an intermediary formula within the "formulas" section of the RASON model.

```
"formulas": {
    "form1": { formula: "LET(x, 1+1, LET(y, 2, A5+x+y))" }
}
```

This function can also be used to formulate an uncertain function in a simulation or stochastic optimization model or an objective function or constraint in an optimization or stochastic/simulation optimization model.

```
 "uncertainFunctions": {
    "uncFunc1": { formula: "LET(x, 1+1, LET(y, 2, A5+x+y))" }
}
```

To see this function used in conjunction with the LAMBDA fuction, see the Box Functions section within the first chapter of this guide. See Microsoft's Office Support for a complete documentation of this function. See below for an example of how to use this function in conjunction with RASON.

Psi Distribution Functions

The PSI Distribution functions are used to define the ‘nature of the uncertainty’ assumed by uncertain variables. They can be broadly classified into four groups:

- *Continuous analytic* distributions such as PsiUniform() and PsiNormal()
- *Discrete analytic* distributions such as PsiBinomial() and PsiGeometric()
- *Custom* distributions such as PsiCumul() and PsiGeneral()
- *Special* distributions such as PsiSip() and PsiSlurp()

On each trial of a simulation, Risk Solver Engine (RSE) draws a *random sample* value from each PSI Distribution function you use. PsiSip() and PsiSlurp() operate differently: On each trial, RSE draws the *next sequential* value listed in the SIP or SLURP for that uncertain variable. Then Risk Solver uses these sample values to calculate your model and its uncertain functions

The sample values drawn for PSI Distribution functions other than PsiSip() and PsiSlurp() depend on the *type* of distribution function, the *parameters* of the distribution (for example, mean and variance for the PsiNormal distribution), and the *property* functions that you pass as additional arguments to the distribution function call, which can shift, truncate, or lock the distribution, or correlate its sample values with samples drawn for other uncertain variables.

To learn more about the analytic probability distributions supported by the RASON modeling language, see the Appendix. , you can consult standard reference texts on probability, statistics, and Monte Carlo simulation, such as *Simulation Modeling and Analysis, 4th Ed.* by Averill Law, *Statistical Distributions, 3rd Ed.* by Merran Evans, Nicholas Hastings and Brian Peacock, *Univariate Discrete Distributions, 3rd Ed.* by Norman Johnson, Adrienne Kemp and Samuel Kotz, or *Continuous Univariate Distributions, Vol. 1 & 2 , 2nd Ed.* by Norman Johnson, Samuel Kotz and N. Balakrishnan.

Continuous Analytic Distributions

All continuous analytic distributions supported in RASON may be found below.

PsiBeta

```
PsiBeta (α1,α2,...)
```

PsiBeta (α_1,α_2) is a flexible distribution for modeling probabilities based on Bayesian statistics. The Beta distribution can be used as an approximation in the absence of specific distribution information. Typical uses include modeling time to complete a task in project networks and Bayesian Statistics.

The Beta distribution can take on a variety of shapes depending on the values of the two parameters α_1 and α_2. The Beta distribution with $\alpha_1 = \alpha_2 = 1$ is the Uniform (0,1) distribution. The Beta distribution with $\alpha_1 = 1$, $\alpha_2 = 2$ is the Left Triangular distribution. The beta distribution with $\alpha_1 = 2$, $\alpha_2 = 1$ is the Right Triangular distribution. A random variable X is defined by PsiBeta (α_1,α_2) if and only if 1 – X is defined by Beta (α_2,α_1).

Parameters

$$\alpha_1, \alpha_2 > 0$$

Range of Function Values

$$[0,1]$$

Probability Density Function

$$f(x) = \begin{cases} \dfrac{x^{\alpha_1 - 1}(1-x)^{\alpha_2 - 1}}{B(\alpha_1, \alpha_2)} \text{if } x \in (0,1) \\ 0 \text{ otherwise} \end{cases}$$

$$B(\alpha_1, \alpha_2) \text{ is the Beta function}$$

$$B(\alpha_1, \alpha_2) = \int_0^1 t^{\alpha_1 - 1} (1-t)^{\alpha_2 - 1} \, dt$$

Cumulative Distribution Function

$$F(x) = \frac{B_x(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}$$

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta Function

$B(\alpha_1, \alpha_2)$ is the Beta Function

Mean

$$\frac{\alpha_1}{\alpha_1 + \alpha_2}$$

Variance

$$\frac{\alpha_1\alpha_2}{(\alpha_1 + \alpha_2)^2(\alpha_1 + \alpha_2 + 1)}$$

Skewness

$$\frac{2(\alpha_2 - \alpha_1)}{\alpha_1 + \alpha_2 + 2}\sqrt{\frac{\alpha_1 + \alpha_2 + 1}{\alpha_1\alpha_2}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1)\left[2(\alpha_1 + \alpha_2)^2 + \alpha_1\alpha_2(\alpha_1 + \alpha_2 - 6)\right]}{\alpha_1\alpha_2(\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)}$$

Median

Not applicable

Mode

$\frac{\alpha_1 - 1}{\alpha_1 + \alpha_2 - 2}$ if $\alpha_1 > 1, \alpha_2 > 1$

0 and 1 if $\alpha_1 < 1, \alpha_2 < 1$

0 if $(\alpha_1 < 1, \alpha_2 \geq 1)$ or if $(\alpha_1 = 1, \alpha_2 > 1)$

1 if $(\alpha_1 \geq 1, \alpha_2 < 1)$ or if $(\alpha_1 > 1, \alpha_2 = 1)$

does not uniquely exist if $\alpha_1 = \alpha_2 = 1$

PsiBetaGen

```
PsiBetaGen (α1,α2,a,b,...)
```

PsiBetaGen (α_1,α_2,a,b) is a *rescaled and relocated* Beta distribution, with lower and upper bounds given respectively by a and b. The shape parameters α_1,α_2 play the same role as in the PsiBeta function. If X is a Beta random variable with support in [0,1], then a + (b – a) X is a Beta random variable with support in [a,b].

Parameters

$$\alpha_1, \alpha_2 > 0$$
$$a < b$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \frac{(x-a)^{\alpha_1 - 1}(b-x)^{\alpha_2 - 1}}{B(\alpha_1, \alpha_2)(b-a)^{\alpha_1 + \alpha_2 - 1}}$$

$B(\alpha_1, \alpha_2)$ is the Beta Function

Cumulative Distribution Function

$$F(x) = \frac{B_z(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}, z = \frac{x-a}{b-a}$$

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta Function

$B(\alpha_1, \alpha_2)$ is the Beta Function

Mean

$$a + \frac{\alpha_1}{\alpha_1 + \alpha_2}(b-a)$$

Variance

$$\frac{\alpha_1 \alpha_2}{(\alpha_1 + \alpha_2)^2 (\alpha_1 + \alpha_2 + 1)}(b-a)^2$$

Skewness

$$\frac{2(\alpha_2 - \alpha_1)}{\alpha_1 + \alpha_2 + 2}\sqrt{\frac{\alpha_1 + \alpha_2 + 1}{\alpha_1 \alpha_2}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1)\left[2(\alpha_1 + \alpha_2)^2 + \alpha_1 \alpha_2 (\alpha_1 + \alpha_2 - 6)\right]}{\alpha_1 \alpha_2 (\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)}$$

Median

Not applicable

Mode

$$a+\frac{\alpha_1-1}{\alpha_1+\alpha_2-2}(b\text{-}a)\text{ if }\alpha_1>1,\alpha_2>1$$

a and b if $\alpha_1<1,\alpha_2<1$

a if $(\alpha_1<1,\alpha_2\geq 1)$ or if $(\alpha_1=1,\alpha_2>1)$

b if $(\alpha_1\geq 1,\alpha_2<1)$ or if $(\alpha_1>1,\alpha_2=1)$

does not uniquely exist if $\alpha_1=\alpha_2=1$

PsiBetaSubj

```
PsiBetaSubj (a,c,µ,b,...)
```

PsiBetaSubj is a flexible distribution like PsiBetaGen, but with parameters you choose for the minimum (a), most likely (c), mean (µ) and maximum (b) values. These parameters are used to compute the shape parameters α_1,α_2 used in the PsiBeta function.

Parameters

$a<\mu<b$

$a<c<b$

$\mu>\frac{a+b}{2}$ if $c>\mu$

$\mu<\frac{a+b}{2}$ if $c<\mu$

$\mu=\frac{a+b}{2}$ if c=μ

The shape parameters α_1,α_2 can be determined using

$$\alpha_2=\alpha_1\frac{b-\mu}{\mu-a}$$

$$\alpha_1=\frac{2(\mu-a)\left(\frac{a+b}{2}-c\right)}{(\mu-c)(b-a)}$$

Range of Function Values

$[a,b]$

Probability Density Function

$$f(x)=\frac{(x-a)^{\alpha_1-1}(b-x)^{\alpha_2-1}}{B(\alpha_1,\alpha_2)(b-a)^{\alpha_1+\alpha_2-1}}$$

$B(\alpha_1,\alpha_2)$ is the Beta Function

Cumulative Distribution Function

$$F(x) = \frac{B_z(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}, z = \frac{x-a}{b-a}$$

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta Function

$B(\alpha_1, \alpha_2)$ is the Beta Function

Mean

μ

Variance

$$\frac{(\mu - a)(b - \mu)(\mu - c)}{a + b + \mu - 3c}$$

Skewness

$$\frac{(a + b - 2\mu)}{\left|\mu + \frac{a+b}{2} - 2c\right|} \sqrt{\frac{(\mu - c)(a + b + \mu - 3c)}{(\mu - a)(b - \mu)}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1)\left[2(\alpha_1 + \alpha_2)^2 + \alpha_1\alpha_2(\alpha_1 + \alpha_2 - 6)\right]}{\alpha_1\alpha_2(\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)}$$

Median

Not applicable

Mode

c

PsiCauchy

```
PsiCauchy (λ,...)
```

PsiCauchy (λ) is a distribution with a central peak, with very heavy tails and no finite moments; it has no moments such as mean, variance, etc. defined, but its mode and median are both equal to zero. The ratio of two independent standard Normal random variables is a Cauchy distribution with parameter λ = 1.

Parameters

$\lambda > 0$

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{1}{\pi\lambda\left[1+\left({x}/{\lambda}\right)^2\right]}$$

Cumulative Distribution Function

$$F(x) = \frac{1}{\pi}\arctan\left(\frac{x}{\lambda}\right)+\frac{1}{2}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

0

Mode

0

PsiChiSquare

```
PsiChiSquare (df,...)
```

PsiChiSquare (df) is a distribution with a finite lower bound of zero, and an infinite upper bound. It is usually used in statistical significance tests.

The Chi Square distribution is a special case of the Gamma distribution. A Chi Square random variable with parameter df = 2 is the same as an Exponential random variable with mean 0.5. As the parameter df approaches infinity, the Chi Square distribution tends to a Normal distribution.

Parameters

$df > 0,\ \text{integer}$

Range of Function Values

$[0,\infty)$

Probability Density Function

$$f(x)=\frac{1}{2^{df/2}\,\Gamma\left(df/2\right)}x^{\left(df/2\right)-1}e^{-x/2}$$

$\Gamma(a)$ is the Gamma Function,

$$\Gamma(a)=\int_0^\infty t^{a-1}e^{-t}dt$$

Cumulative Distribution Function

$$F(x)=\frac{\gamma\left(df/2,x/2\right)}{\Gamma\left(df/2\right)}$$

$\Gamma(a)$ is the Gamma Function,

$\gamma(a,b)$ is the Incomplete Gamma Function

Mean

df

Variance

$2df$

Skewness

$$\sqrt{8/df}$$

Kurtosis

$$\frac{12}{df}+3$$

Median

$$df-2/3$$

Mode

$$\begin{cases}(df-2)\text{ if }df\geq 2\\ 0\text{ if }df=1\end{cases}$$

PsiErf

```
PsiErf (h,...)
```

PsiErf is a distribution based on the “error function” ERF(x). Its shape is closely related to the Normal distribution.

Parameters

$h>0$

Range of Function Values

$$(-\infty,\infty)$$

Probability Density Function

$$f(x)=\frac{h}{\sqrt{\pi}}e^{-(hx)^2}$$

Cumulative Distribution Function

$$F(x)=\Phi\left(\sqrt{2}hx\right)$$

$\Phi(v)$ is the Error Function

$$\Phi(v)=\frac{2}{\sqrt{\pi}}\int_0^v e^{-t^2}dt$$

Mean

0

Variance

$$\frac{1}{2h^2}$$

Skewness

0

Kurtosis

3

Median

0

Mode

0

PsiErlang

```
PsiErlang (k,β,...)
```

PsiErlang (k,β) is a distribution with a finite lower bound, closely related to the Gamma and Exponential distributions. It has applications in reliability and queuing models. When the parameter k = 1, the Erlang distribution is the same as an Exponential distribution.

Parameters

$k,\beta>0$

k integer

Range of Function Values

$$[0,\infty)$$

Probability Density Function

$$f(x) = \frac{x^{k-1} e^{-\left(x/\beta\right)}}{\beta^k (k-1)!}$$

Cumulative Distribution Function

$$F(x) = \frac{\gamma\left(k, \frac{x}{\beta}\right)}{(k-1)!}$$

$\gamma(x, y)$ is the Incomplete Gamma Function

Mean

$$k\beta$$

Variance

$$k\beta^2$$

Skewness

$$\frac{2}{\sqrt{k}}$$

Kurtosis

$$3 + \frac{6}{k}$$

Median

Not defined

Mode

$$\beta(k-1)$$

PsiExponential

`PsiExponential (β,...)`

PsiExponential (β) is a distribution with a finite lower bound and rapidly decreasing values. It can be used to represent time between random occurrences in queuing and reliability engineering applications.

The minimum of a set of independent exponential random variables is also an exponentially distributed random variable.

Parameters

$$\beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f\left(x\right)=\frac{1}{\beta}e^{-x/\beta}$$

Cumulative Distribution Function

$$F\left(x\right)=\begin{cases}1-e^{-x/\beta} & \text{if } x\geq 0\\ 0 \text{ otherwise}\end{cases}$$

Mean

β

Variance

β^2

Skewness

2

Kurtosis

9

Median

$\beta\ln\left(2\right)$

Mode

0

PsiGamma

`PsiGamma (α,β,...)`

PsiGamma (α,β) is a flexible distribution with a finite lower bound and decreasing values. PsiExponential, PsiErlang, and PsiChiSquare are special cases of PsiGamma, as explained below. The Gamma distribution is often used to model the time between events that occur with a constant average rate.

When $\alpha = 1$, the Gamma distribution is the same as an Exponential distribution. If the parameter α is integer, then the Gamma distribution is the same as the Erlang distribution. The Gamma distribution with $\alpha = a/2$, $\beta = 2$ is the same as a Chi Square distribution with parameter a (a degrees of freedom).

If X_1, $X_{2,\ldots}X_m$ are independent random variables with X_i ~ PsiGamma (α_i,β), then their sum also has a Gamma distribution with parameters ($\alpha_1 + \alpha_2 + \ldots + \alpha_m$,β). Additionally, the Gamma distribution approaches a normal distribution with the same mean and standard deviation as the parameter α approaches infinity.

Parameters

$\alpha,\beta>0$

Range of Function Values

$\left[0,\infty\right)$

Probability Density Function

$$f(x)=\begin{cases}\dfrac{\beta^{-\alpha}x^{\alpha-1}e^{-x/\beta}}{\Gamma(\alpha)} & \text{if } x>0\\ 0 \text{ otherwise}\end{cases}$$

$\Gamma(\alpha)$ is the Gamma Function

Cumulative Distribution Function

$$F(x)=\frac{\gamma\left(\alpha, x/\beta\right)}{\Gamma(\alpha)}$$

$\gamma(a,b)$ is the Incomplete Gamma Function

Mean

$$\beta\alpha$$

Variance

$$\beta^2\alpha$$

Skewness

$$\frac{2}{\sqrt{\alpha}}$$

Kurtosis

$$\frac{6}{\alpha}+3$$

Median

Not defined

Mode

$$\begin{cases}\beta(1-\alpha) \text{ if } \alpha\geq 1\\ 0 \text{ otherwise}\end{cases}$$

PsiInvNormal

```
PsiInvNormal (μ,λ,...)
```

PsiInvNormal (μ,λ) is a distribution with a finite lower bound, where it is zero. The Inverse Normal distribution is used to model Brownian motion and other diffusion processes. As the parameter λ tends to infinity, the Inverse Normal distribution approaches a Normal distribution.

Parameters

$$\mu,\lambda>0$$

Range of Function Values

$$(0,\infty)$$

Probability Density Function

$$f(x)=\sqrt{\frac{\lambda}{2\pi x^3}}\left[e^{\left(\frac{-\lambda(x-\mu)^2}{2x\mu^2}\right)}\right]$$

Cumulative Distribution Function

$$F(x)=\Phi\left(\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu}-1\right)\right)+e^{\left(\frac{2\lambda}{\mu}\right)}\Phi\left(-\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu}+1\right)\right)$$

$\Phi(z)$ is the Error Function

Mean

$$\mu$$

Variance

$$\frac{\mu^3}{\lambda}$$

Skewness

$$3\sqrt{\frac{\mu}{\lambda}}$$

Kurtosis

$$15\frac{\mu}{\lambda}+3$$

Median

Not defined

Mode

$$\mu\left\{\left(1+\frac{9\mu^2}{4\lambda^2}\right)^{1/2}-\frac{3\mu}{2\lambda}\right\}$$

PsiLaplace

```
PsiLaplace (β,...)
```

PsiLaplace (β) is an unbounded, fat-tailed distribution that describes the difference between two independent exponentials. If a random variable X has a Laplace distribution, then |X| has an Exponential distribution.

Parameters

$$\beta > 0$$

Range of Function Values

$$(-\infty,\infty)$$

Probability Density Function

$$f(x)=\frac{e^{-\left(|x|/\beta\right)}}{2\beta}$$

Cumulative Distribution Function

$$F(x)=\begin{cases}1-\frac{1}{2}e^{-x/\beta} & \text{if } x\geq 0\\ \frac{1}{2}e^{x/\beta} & \text{otherwise}\end{cases}$$

Mean

0

Variance

$$2\beta^2$$

Skewness

0

Kurtosis

3

Median

0

Mode

0

PsiLogistic

```
PsiLogistic (µ,s,...)
```

PsiLogistic (µ,s) is an unbounded distribution, symmetric around its mean, with broader tails than the Normal distribution. The Logistic distribution is often used to model growth processes.

Parameters

$$\mu$$

$$s>0$$

Range of Function Values

$$(-\infty,\infty)$$

Probability Density Function

$$f(x)=\frac{e^{-(x-\mu)/s}}{s\left(1+e^{-(x-\mu)/s}\right)^2}$$

Cumulative Distribution Function

$$F(x) = \frac{1}{1 + e^{-(x-\mu)/s}}$$

Mean

$$\mu$$

Variance

$$\frac{\pi^2 s^2}{3}$$

Skewness

0

Kurtosis

6/5

Median

$$\mu$$

Mode

$$\mu$$

PsiLogLogistic

`PsiLogLogistic (γ,β,α,...)`

PsiLogLogistic (γ,β,α) is a distribution with a finite lower bound. The natural log of PsiLogLogistic is a Logistic random variable. The Log-Logistic distribution can be used to model the time to perform a job or task.

Parameters

$$\beta, \alpha > 0$$

$$\gamma$$

Range of Function Values

$$[\gamma, \infty)$$

Probability Density Function

$$f(x) = \frac{\alpha \left(\frac{x-\gamma}{\beta} \right)^{\alpha-1}}{\beta \left[1 + \left(\frac{x-\gamma}{\beta} \right)^{\alpha} \right]^2}$$

Cumulative Distribution Function

$$F\left(x\right)=\frac{1}{1+\left(\frac{\beta}{x-\gamma}\right)^{\alpha}}$$

Mean

$$\frac{\beta\pi\cos ec\left(\frac{\pi}{\alpha}\right)}{\alpha}+\gamma \text{ for } \alpha>1$$

Variance

$$\frac{\beta^{2}\pi\left[2\cos ec\left(\frac{2\pi}{\alpha}\right)-\frac{\pi}{\alpha}\cos ec^{2}\left(\frac{\pi}{\alpha}\right)\right]}{\alpha} \text{ for } \alpha>2$$

Skewness

$$\frac{\left[3\text{cosec}\left(\frac{3\pi}{\alpha}\right)-\frac{6\pi}{\alpha}\cos ec\left(\frac{2\pi}{\alpha}\right)\cos ec\left(\frac{\pi}{\alpha}\right)+\frac{2\pi^{2}}{\alpha^{2}}\cos ec^{3}\left(\frac{\pi}{\alpha}\right)\right]}{\left(\sqrt{\frac{\pi}{\alpha}}\right)\left[2\cos ec\left(\frac{2\pi}{\alpha}\right)-\frac{\pi}{\alpha}\cos ec^{2}\left(\frac{\pi}{\alpha}\right)\right]^{3/2}} \text{ for } \alpha>3$$

Kurtosis

Not defined

Median

Not defined

Mode

$$\begin{cases}\gamma+\beta\left[\frac{\alpha-1}{\alpha+1}\right]^{1/\alpha} & \text{for } \alpha>1\\ 0 \text{ otherwise}\end{cases}$$

PsiLogNormal

```
PsiLogNormal (μ,σ,...)
```

PsiLogNormal (μ,σ) is a distribution with a finite lower bound and has mean μ and standard deviation σ. The LogNormal distribution can be used to model quantities that are products of many small independent variables. The natural log of PsiLogNormal is a Normal random variable.

Parameters

$$\mu,\sigma>0$$

Range of Function Values

$$\left[0,\infty\right)$$

Probability Density Function

$$f(x)=\frac{e^{-\frac{1}{2}\left(\frac{\ln x-\mu'}{\sigma'}\right)^2}}{x\sqrt{2\pi}\sigma'}$$

$$\sigma'=\sqrt{\ln\left(1+\frac{\sigma^2}{\mu^2}\right)},\mu'=\ln\left(\frac{\mu^2}{\sqrt{\mu^2+\sigma^2}}\right)$$

Cumulative Distribution Function

$$F(x)=\Phi\left(\frac{\ln x-\mu'}{\sigma'}\right)$$

$\Phi(a)$ is the Error Function

Mean

$$\mu$$

Variance

$$\sigma^2$$

Skewness

$$\frac{\sigma^3}{\mu^3}+\frac{3\sigma}{\mu}$$

Kurtosis

$$6\left(1+\frac{\sigma}{\mu}\right)^4+2\left(1+\frac{\sigma}{\mu}\right)^3+3\left(1+\frac{\sigma}{\mu}\right)^2-3$$

Median

$$\frac{\mu^2}{\sqrt{\mu^2+\sigma^2}}$$

Mode

$$\frac{\mu^4}{\left(\sigma^2+\mu^2\right)^{3/2}}$$

PsiLogNorm2

`PsiLogNorm2 (μ,σ,...)`

PsiLogNorm2 (μ,σ) is a distribution with a finite lower bound. It can be used to model quantities that are products of many small independent variables. The natural log of PsiLogNorm2 is a Normal random variable. In contrast to PsiLogNormal(), the parameters μ and σ of PsiLogNorm2() are the mean and standard deviation of the *corresponding Normal distribution.*

Parameters

μ

$\sigma > 0$

Range of Function Values

$\left[0,\infty\right)$

Probability Density Function

$$f\left(x\right) = \frac{e^{-\frac{1}{2}\left(\frac{\ln x-\mu}{\sigma}\right)^2}}{x\sqrt{2\pi}\sigma}$$

Cumulative Distribution Function

$$F\left(x\right) = \Phi\left(\frac{\ln x-\mu}{\sigma}\right)$$

$\Phi\left(a\right)$ is the Error Function

Mean

$e^{\mu+\sigma^2/2}$

Variance

$\left(e^{\sigma^2}-1\right)e^{2\mu+\sigma^2}$

Skewness

$\left(e^{\sigma^2}+2\right)\sqrt{e^{\sigma^2}-1}$

Kurtosis

${}_6e^{4\sigma^2}+2e^{3\sigma^2}+3e^{2\sigma^2}-3$

Median

e^{μ}

Mode

$e^{\mu-\sigma^2}$

PsiMaxExtreme

`PsiMaxExtreme (m,s,...)`

PsiMaxExtreme (m,s) is the positively skewed form of the Extreme Value distribution, which is the limiting distribution of a very large collection of random observations.

Parameters

m

$s > 0$

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{z}{s} e^{-z}, z = e^{\frac{-(x-m)}{s}}$$

Cumulative Distribution Function

Not defined

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiMinExtreme

`PsiMinExtreme (m,s,...)`

PsiMinExtreme is the negatively skewed form of the Extreme Value distribution, which is the limiting distribution of a very large collection of random observations.

Parameters

m

$s > 0$

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{z}{s} e^{-z}, z = e^{\frac{(x-m)}{s}}$$

Cumulative Distribution Function

Not defined

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiMyerson

```
PsiMyerson (a,b,c,t,...)
```

PsiMyerson (a,b,c,t) is a generalized LogNormal/Normal distribution, specified using the bottom percentile (a), 50th percentile (b), top percentile (c) and optional tail percentage parameter (t). This distribution is bounded on the side of the narrower percentile range; when both the bottom and top percentile ranges are equal, then this distribution is unbounded.

If the t parameter (tail percentage) is present then the a and c parameters (bottom and top percentiles) are used to construct a distribution PDF in such a way that the left and right tails (remaining equal) sum up to the desired t parameter value. The top percentile is *always* symmetric to the bottom percentile. For example, if the bottom percentile equals the 20th percentile, the top percentile will be equal to the 80th percentile.

The default option for parameter t is 0.50 which means that the left tail and the right tail each equal 0.25. As a result, parameter a (bottom percentile) is the 25th percentile and parameter c (top percentile) is the 75th percentile.

This distribution, developed by Dr. Roger Myerson, is used to model random variables when the only information available is the percentile values, and optionally, a tail percentage parameter indicating the probability of values being within the specified percentiles. If the specified percentiles are equidistant (measured by the parameter b' below), then the Myerson distribution is equivalent to a Normal distribution. When the 50th percentile is equal to the geometric mean of the top and bottom percentiles, then the Myerson distribution is equivalent to the LogNormal distribution.

Parameters

$a < c < b$

$t \in (0,1)$

If t is omitted, it is given a default value of 0.5

Range of Function Values

$[LB, UB)$

where,

$LB = -\infty, UB = \infty$ if $b' = 1$

$LB = c - \frac{(b-c)}{b'-1}, UB = \infty$ if $b' > 1$

$LB = -\infty, UB = c - \frac{(b-c)}{b'-1}$ if $b' < 1$

and,

$$b' = \frac{b-c}{c-a}$$

Probability Density Function

If $b' \neq 1$,

$$f(x) = \frac{z_{\left(1-t/2\right)}(b'-1)}{\left((b-c)+(x-c)(b'-1)\right)\ln(b')} f_{N(0,1)}(q)$$

where

$f_{N(0,1)}(q)$ is the PDF of the Standard Normal distribution,

$$\mathrm{q} = \mathrm{Z}_{\left(1\text{-}\mathrm{t}/2\right)}\left\{\frac{\ln\left(\frac{(x-c)(b'-1)}{(b-c)}+1\right)}{\ln(b')}\right\}$$

If $b' = 1$,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

where

$\mu = c$

$$\sigma = \frac{(b-c)}{Z_{\left(1-t/2\right)}}$$

and

$\mathrm{Z}_{\mathrm{x}} =$ CDFInverse of the Standard Normal distribution at x

Cumulative Distribution Function

If $b' \neq 1$,

$$F(x) = F_{N(0,1)}(q) = \frac{1}{2}\left(\Phi\left(\frac{q}{\sqrt{2}}\right) + 1\right)$$

where

$F_{N(0,1)}(q)$ is the CDF of the Standard Normal distribution

$\Phi(x) = \frac{2}{\sqrt{\pi}}\int_{0}^{x} e^{-t^2}dt$ is the Error function

$$\text{q} = \text{Z}_{\left(1\text{-}t/2\right)}\left\{\frac{\ln\left(\frac{(x-c)(b'-1)}{(b-c)}+1\right)}{\ln(b')}\right\}$$

If $b' = 1$,

$$F(x) = \frac{1}{2}\left[1 + erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

where

$erf(y)$ is the Error Function, and

$\mu = c$

$$\sigma = \frac{(b-c)}{Z_{\left(1-t/2\right)}}$$

and

$\text{Z}_\text{x} = \text{CDFInverse of the Standard Normal distribution at x}$

Mean

No closed form

Variance

No closed form

Skewness

No closed form

Kurtosis

No closed form

Median

No closed form

Mode

No closed form

PsiNormal

```
PsiNormal (μ,σ,...)
```

PsiNormal (μ,σ) is an unbounded, symmetric distribution with the familiar *bell curve*, also called a Gaussian distribution. The Normal distribution is widely used in many different kinds of applications. A normal distribution with mean zero and standard deviation one is called a *Standard* normal distribution.

The sum of independent random variables of any shape tends to the Normal distribution.

Parameters

μ

$\sigma > 0$

Range of Function Values

$\left(-\infty,\infty\right)$

Probability Density Function

$$f\left(x\right)=\frac{1}{\sigma\sqrt{2\pi}}e^{\left(-\frac{\left(x-\mu\right)^2}{2\sigma^2}\right)}$$

Cumulative Distribution Function

$$F\left(x\right)=\frac{1}{2}\left[1+erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

$erf\left(y\right)$ is the Error Function

Mean

μ

Variance

σ^2

Skewness

0

Kurtosis

0

Median

μ

Mode

μ

PsiNormalSkew

```
PsiNormalSkew(a,b,c,...)
```

PsiNormalSkew (a,b,c) is a generalized Normal distribution with lower bound a, upper bound b, and skew value c. Lower and upper bound values describe +/- 3rd standard deviation. The skew value c can take on values between (but not including) -1 and 1. While the Normal distribution is symmetric, the Normal Skew distribution is skewed either to the left with a positive skew parameter or to the right with a negative skew parameter.

Both the Myerson distribution (described above) and the PsiNormalSkew distribution have recently emerged in practice. Both distributions are generalizations of the Normal distribution but rather than using the mean and standard deviation as arguments, these distributions are calculated using an *upper* and *lower* bound along with either *likely* and *tail* arguments (such as with the Myerson distribution) or a skew argument (such as with the NormalSkew distribution). When the skew argument is equidistant from the upper and lower bounds, the NormalSkew distribution equals the Myerson distribution.

In the PsiNormalSkew distribution, the lower and upper bounds are exactly the same as in the Myerson distribution. The tail argument is missing in the Normal Skew distribution as it remains at the constant value of 0.002699796146511.

Parameters

a < b

-1< c <1

If *c* is omitted, it is given a default value of 0. In this case, the PsiNormalSkew distribution will equal the PsiMyerson distribution.

Range of Function Values

$$[LB, UB]$$

$$where,$$

$$LB = -\infty, UB = \infty \, if \; b' = 1$$

$$LB = d - \frac{(b-d)}{b'-1}, UB = \infty \, if \; b' > 1$$

$$LB = -\infty, UB = d - \frac{(b-d)}{b'-1} \, if \; b' < 1$$

$$where$$

$$b' = \frac{b-d}{d-a}$$

$$and$$

$$d = a + (c+1) * \frac{(b-1)}{2}$$

$$(The \, "d" \, parameter \, here \, equals \, the \, "c" \, parameter \, in \, the \, PsiMyerson \, distribution.)$$

Probability Density Function

$If\ b' \neq 1,$

$$f(x) = \frac{z_{(1-\frac{t}{2})}(b'-1)}{((b-d)+(\mathrm{x-d})(\mathrm{b'}-1))\ln(\mathrm{b'})}\, f_{N(0,1)}(q)$$

where

$\mathrm{f}_{\mathrm{N}(0,1)}(\mathrm{q})$ is the PDF of the Standard Normal distribution.

$$\mathrm{q} = Z_{(1-\frac{t}{2})}\left\{\frac{\ln\left(\frac{(x-d)(b'-1)}{(b-d)}+1\right)}{\ln(b')}\right\}$$

$If\ b' = 1,$

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{\left(\frac{(\chi-\mu)^2}{2\sigma^2}\right)}$$

where

$\mu = d$

$$\sigma = \frac{(b-d)}{Z_{(1-\frac{t}{2})}}$$

$$d = a + (c+1) * \frac{(b-1)}{2}$$

(The "d" parameter here equals the "c" parameter in the PsiMyerson distribution.)

and

$Z_\chi = CDFInverse\ of\ the\ Normal\ distribution\ at\ \chi.$

Cumulative Distribution Function

If $b' \neq 1$,

$$F\left(x\right) = F_{N\left(0,1\right)}\left(q\right) = \frac{1}{2}\left(\Phi\left(\frac{q}{\sqrt{2}}\right)+1\right)$$

where

$F_{N\left(0,1\right)}\left(q\right)$ is the CDF of the Standard Normal distribution

$\Phi\left(x\right) = \frac{2}{\sqrt{\pi}}\int_{0}^{x} e^{-t^2}\,dt$ is the Error function

$$q = Z_{\left(1\text{-}t/2\right)}\left\{\frac{\ln\left(\frac{\left(x-d\right)\left(b'-1\right)}{\left(b-d\right)}+1\right)}{\ln\left(b'\right)}\right\}$$

If $b' = 1$,

$$F\left(x\right) = \frac{1}{2}\left[1+erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

where

$erf\left(y\right)$ is the Error Function, and

$\mu = d$

$$\sigma = \frac{\left(b-d\right)}{Z_{\left(1-t/2\right)}}$$

$$d = a + (c+1) * \frac{(b-1)}{2}$$

$(The\text{"}d\text{"}\ parameter\ here\ equals\ the\text{"}c\text{"}\ parameter\ in\ the\ PsiMyerson\ distribution.)$

and

Z_{x} = CDFInverse of the Standard Normal distribution at x

Mean

No closed form

Variance

No closed form

Skewness

No closed form

Kurtosis

No closed form

Median

No closed form

Mode

No closed form

PsiPareto

`PsiPareto (θ,a,...)`

PsiPareto (θ,a) is a distribution with a finite lower bound a, and shape parameter θ. The Pareto distribution can be used to describe or model wealth distribution, sizes of particles, etc. It is the exponential of an Exponential random variable.

Parameters

$\theta, a > 0$

Range of Function Values

$\left[a, \infty\right)$

Probability Density Function

$$f\left(x\right) = \frac{\theta a^{\theta}}{x^{\theta+1}}$$

Cumulative Distribution Function

$$F\left(x\right) = 1 - \left(\frac{a}{x}\right)^{\theta}$$

Mean

$$\frac{a\theta}{\theta - 1} \text{ for } \theta > 1$$

Variance

$$\frac{\theta a^2}{\left(\theta - 1\right)^2 \left(\theta - 2\right)} \text{ for } \theta > 2$$

Skewness

$$\frac{2\left(\theta + 1\right)}{\left(\theta - 3\right)} \sqrt{\frac{\left(\theta - 2\right)}{\theta}} \text{ for } \theta > 3$$

Kurtosis

$$\frac{3\left(\theta - 2\right)\left(3\theta^2 + \theta + 2\right)}{\theta\left(\theta - 3\right)\left(\theta - 4\right)} \text{ for } \theta > 4$$

Median

$$a2^{1/\theta}$$

Mode

a

PsiPareto2

```
PsiPareto2 (b,q,...)
```

PsiPareto2 is an alternate form of the Pareto distribution with a finite lower bound of 0, and a shape parameter q. Like PsiPareto (θ,a), it can be used to describe or model wealth distribution, sizes of particles, etc. It is the exponential of an Exponential random variable.

Parameters

$b, q > 0$

Range of Function Values

$[0, \infty)$

Probability Density Function

$$f(x) = \frac{qb^q}{(x+b)^{q+1}}$$

Cumulative Distribution Function

$$F(x) = 1 - \left(\frac{b}{x+b}\right)^q$$

Mean

$$\frac{b}{q-1} \text{ for q} > 1$$

Variance

$$\frac{qb^2}{(q-1)^2(q-2)} \text{ for q} > 2$$

Skewness

$$\frac{2(q+1)}{(q-3)}\sqrt{\frac{(q-2)}{q}} \text{ for q} > 3$$

Kurtosis

Not defined

Median

$$\frac{b}{q}2^{1/q}$$

Mode

0

PsiPearson5

```
PsiPearson5 (α,β,...)
```

PsiPearson5 (α,β) is a distribution with a lower bound of 0, and density similar to that of the LogNormal distribution. The Pearson5 distribution is sometimes called the Inverse Gamma distribution. It can be used to model time delays when these can possibly take on unbounded (or very large) values.

Parameters

$\alpha, \beta > 0$

Range of Function Values

$[0, \infty)$

Probability Density Function

$$f(x) = \frac{x^{-(\alpha+1)} e^{-\beta/x}}{\beta^{-\alpha}\Gamma(\alpha)}$$

Cumulative Distribution Function

$$F(x) = 1 - F_G\left(\frac{1}{x}\right)$$

$F_G(y)$ is the Distribution function of a Gamma $\left(\alpha, 1/\beta\right)$ random variable

Mean

$\frac{\beta}{\alpha-1}$ for $\alpha > 1$

Variance

$\frac{\beta^2}{(\alpha-1)^2(\alpha-2)}$ for $\alpha > 2$

Skewness

$\frac{4\sqrt{\alpha-2}}{\alpha-3}$ for $\alpha > 3$

Kurtosis

$\frac{3(\alpha+5)(\alpha-2)}{(\alpha-3)(\alpha-4)}$ for $\alpha > 4$

Median

Not defined

Mode

$$\frac{\beta}{\alpha+1}$$

PsiPearson6

```
PsiPearson6 (α1,α2,β,...)
```

PsiPearson6 (α_1, α_2, β) is a distribution with a lower bound of 0, and a mode just beyond the lower bound. The Pearson6 distribution is sometimes called the Beta distribution of the second kind.

If X_1 ~ Gamma (α_1,β) and X_2 ~ Gamma (α_2,1) are independent random variables, then X_1/X_2 has a Pearson6 distribution. If X is a random variable with a Pearson6 (α_1,α_2,1) distribution, then X/(1+X) has a Beta (α_1, α_2) distribution.

Parameters

$$\alpha_1,\alpha_2,\beta>0$$

Range of Function Values

$$[0,\infty)$$

Probability Density Function

$$f(x)=\frac{\left(x/\beta\right)^{\alpha_1-1}}{\beta B(\alpha_1,\alpha_2)\left[1+\left(x/\beta\right)\right]^{\alpha_1+\alpha_2}}$$

$B(\alpha_1,\alpha_2)$ is the Beta function

Cumulative Distribution Function

$$F(x)=F_B\left(\frac{x}{x+\beta}\right)$$

$F_B(y)$ is the Distribution function of a Beta (α_1,α_2) random variable

Mean

$$\frac{\beta\alpha_1}{\alpha_2-1} \text{ for } \alpha_2>1$$

Variance

$$\frac{\beta^2\alpha_1(\alpha_1+\alpha_{2-1})}{(\alpha_2-1)^2(\alpha_2-2)} \text{ for } \alpha_2>2$$

Skewness

$$\sqrt{\frac{\alpha_2 - 2}{\alpha_1(\alpha_1 + \alpha_2 - 1)}}\left(\frac{4\alpha_1 + 2\alpha_2 - 2}{\alpha_2 - 3}\right) \text{ for } \alpha_2 > 3$$

Kurtosis

$$\frac{3(\alpha_2 - 2)}{(\alpha_2 - 3)(\alpha_2 - 4)}\left[\frac{2(\alpha_2 - 1)^2 + \alpha_1(\alpha_1 + \alpha_2 - 1)(\alpha_2 + 5)}{\alpha_1(\alpha_1 + \alpha_2 - 1)}\right] \text{ for } \alpha_2 > 4$$

Median

Not defined

Mode

$$\begin{cases} \dfrac{\beta(\alpha_1 - 1)}{\alpha_2 + 1} \text{ if } \alpha_1 \geq 1 \\ 0 \text{ otherwise} \end{cases}$$

PsiPert

```
PsiPert (a,c,b,...)
```

PsiPert (a,c,b) is a form of the Beta distribution, often used to estimate project completion times in the Program Evaluation and Review Technique, where a is the minimum time, b is the maximum time, and c is the most likely time. These parameters are used to compute the shape parameters α_1,α_2 used in the PsiBeta function, as shown below.

Parameters

$a < c < b$

The shape parameters α_1, α_2 can be defined as

$$\alpha_1 = \frac{-5a + b + 4c}{6(b - a)}$$

$$\alpha_2 = \frac{-a + 5b - 4c}{6(b - a)}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \frac{(x - a)^{\alpha_1 - 1}(b - x)^{\alpha_2 - 1}}{B(\alpha_1, \alpha_2)(b - a)^{\alpha_1 + \alpha_2 - 1}}$$

$B(\alpha_1, \alpha_2)$ is the Beta function

Cumulative Distribution Function

$$F\left(x\right)=\frac{B_{\left(\frac{x-a}{b-a}\right)}\left(\alpha_1,\alpha_2\right)}{B\left(\alpha_1,\alpha_2\right)}$$

$B\left(\alpha_1,\alpha_2\right)$ is the Beta function

$B_x\left(\alpha_1,\alpha_2\right)$ is the Incomplete Beta function

Mean

$$\frac{a+b+4c}{6}$$

Variance

$$\frac{\left(b-a\right)^2\alpha_1\alpha_2}{252}$$

Skewness

$$\frac{a+b-4c}{\left(b-a\right)}\sqrt{\frac{7}{\alpha_1\alpha_2}}$$

Kurtosis

$$\frac{3\left(\alpha_1+\alpha_2+1\right)\left[2\left(\alpha_1+\alpha_2\right)^2+\alpha_1\alpha_2\left(\alpha_1+\alpha_2-6\right)\right]}{\alpha_1\alpha_2\left(\alpha_1+\alpha_2+3\right)\left(\alpha_1+\alpha_2+2\right)}$$

Median

Not defined

Mode

C

PsiRayleigh

```
PsiRayleigh (β,...)
```

PsiRayleigh (β) is a distribution with a finite lower bound of 0, a special case of a Weibull distribution. The Rayleigh distribution can be used to model component lifetimes.

If X is a random variable with Rayleigh distribution with parameter β = 1, then X^2 has a Chi Square distribution with parameter 2 (two degrees of freedom). If X and Y are independent normally distributed random variables with mean zero and variance σ^2, then $(X^2+Y^2)^{1/2}$ has a Rayleigh distribution with parameter σ. Thus, a Rayleigh distribution may be used to model the length of a two-dimensional vector whose components are independent and normally distributed.

Parameters

$$\beta>0$$

Range of Function Values

$$\left[0,\infty\right)$$

Probability Density Function

$$f(x) = \frac{xe^{\left(-x^2 / 2\beta^2\right)}}{\beta^2}$$

Cumulative Distribution Function

$$F(x) = 1 - e^{\left(-x^2 / 2\beta^2\right)}$$

Mean

$$\beta\sqrt{\frac{\pi}{2}}$$

Variance

$$\frac{(4-\pi)\beta^2}{2}$$

Skewness

$$\frac{2\sqrt{\pi}(\pi-3)}{(4-\pi)^{3/2}}$$

Kurtosis

$$\frac{-3\pi^2+32}{(4-\pi)^2}$$

Median

$$\beta\sqrt{\ln(4)}$$

Mode

$$\beta$$

PsiStudent

```
PsiStudent (df,…)
```

PsiStudent (df) is an unbounded distribution, symmetric about zero, with a shape similar to that of a Standard Normal distribution, and it approaches the Standard Normal distribution as the degrees of freedom (parameter df) increases. It is also known as the t-distribution, or Student’s t-distribution.

The Student or t-distribution frequently arises when estimating the mean of a normally distributed population when the sample size is small. It is also used when the population variance is unknown, and is estimated from a small sample.

Parameters

$$df > 0,\ \text{integer}$$

Range of Function Values

$$(-\infty,\infty)$$

Probability Density Function

$$f\left(x\right)=\frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df}\,\Gamma\left(\frac{df}{2}\right)}\left(\frac{df}{x^2+df}\right)^{\frac{df+1}{2}}$$

$\Gamma\left(x\right)$ is the Gamma function

Cumulative Distribution Function

$$F\left(x\right)=\frac{1+B_{\left(\frac{x^2}{x^2+df}\right)}\left(\frac{1}{2},\frac{df}{2}\right)}{2}$$

$B_a\left(x,y\right)$ is the Incomplete Beta function

Mean

0 if df > 1
undefined if df = 1

Variance

$$\frac{df}{df-2}\text{ if }df>2$$

Skewness

0 if $df>3$

Kurtosis

$$\frac{3df-6}{df\text{ - }4}\text{ if }df>4$$

Median

0

Mode

0

PsiTriangGen

`PsiTriangGen (a`$_p$`,m,b`$_r$`,p,r…)`

PsiTriangGen (a_p,m,b_r,p,r…) is a Triangular distribution where the lower and upper bounds are not given as fixed values, but are specified using percentiles. This distribution is usually used to create rough models in situations where little or no data is available. The distribution has a most likely value of m, the p (lower) percentile value is a_p, and the r (upper) percentile value is b_r. Given these values, a PsiTriangGen (a_p,m,b_r,p,r…)

distribution corresponds to a PsiTriangular (a,c,b) distribution with values for the bounds (a and b), and the most likely value (c) computed as shown below.

Parameters

$$p, r \in (0,1)$$

$$p < r$$

$$a_p < b_r$$

$$a_p \le m \le b_r$$

The parameters of the Triangular distribution are defined as

$$a = \frac{a_p - m\sqrt{\frac{p}{q}}}{1 - \sqrt{\frac{p}{q}}}$$

$$c = m$$

$$b = \frac{b_r - m\sqrt{\frac{1-r}{1-q}}}{1 - \sqrt{\frac{1-r}{1-q}}}$$

Here q is a solution to the following equation

$$q = \frac{(m - a_p)\left(1 - \sqrt{\frac{1-r}{1-q}}\right)}{(b_r - m)\left(1 - \sqrt{\frac{p}{q}}\right) + (m - a_p)\left(1 - \sqrt{\frac{1-r}{1-q}}\right)}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & \text{if } a \le x \le c \\ \frac{2(b-x)}{(b-c)(b-a)} & \text{if } c \le x \le b \end{cases}$$

Cumulative Distribution Function

$$F(x)=\begin{cases}\dfrac{(x-a)^2}{(c-a)(b-a)} & \text{if } a\le x\le c\\ 1-\dfrac{(b-x)^2}{(b-c)(b-a)} & \text{if } c\le x\le b\end{cases}$$

Mean

$$\frac{a+b+c}{3}$$

Variance

$$\frac{a^2+b^2+c^2-ab-ac-bc}{18}$$

Skewness

$$\frac{\sqrt{2}(a+b-2c)(2a-b-c)(a-2b+c)}{5\left(a^2+b^2+c^2-ab-ac-bc\right)^{3/2}}$$

Kurtosis

12/5

Median

$$\begin{cases}a+\sqrt{\dfrac{(b-a)(c-a)}{2}} & \text{if } c\ge\dfrac{b-a}{2}\\ b-\sqrt{\dfrac{(b-a)(b-c)}{2}} & \text{otherwise}\end{cases}$$

Mode

c

PsiTriangular

```
PsiTriangular (a,c,b,...)
```

PsiTriangular (a,c,b) is a distribution with lower bound a, upper bound b, and most likely value c. This distribution is usually used to create rough models in situations where little or no data is available. If the parameter c = b, then the distribution is also known as a Right Triangular distribution. If the parameter c = a, then the distribution is also known as a Left Triangular distribution. If X_1 and X_2 are independent Uniform (0,1) random variables, then $(X_1+X_2)/2$ has a Triangular (0,0.5,1) distribution.

Parameters

$a\le c\le b$

$a<b$

Range of Function Values

$$[a,b]$$

Probability Density Function

$$f(x)=\begin{cases}\dfrac{2(x-a)}{(c-a)(b-a)} & \text{if } a\le x\le c\\[2ex] \dfrac{2(b-x)}{(b-c)(b-a)} & \text{if } c\le x\le b\end{cases}$$

Cumulative Distribution Function

$$F(x)=\begin{cases}\dfrac{(x-a)^2}{(c-a)(b-a)} & \text{if } a\le x\le c\\[2ex] 1-\dfrac{(b-x)^2}{(b-c)(b-a)} & \text{if } c\le x\le b\end{cases}$$

Mean

$$\frac{a+b+c}{3}$$

Variance

$$\frac{a^2+b^2+c^2-ab-ac-bc}{18}$$

Skewness

$$\frac{\sqrt{2}(a+b-2c)(2a-b-c)(a-2b+c)}{5\left(a^2+b^2+c^2-ab-ac-bc\right)^{3/2}}$$

Kurtosis

12/5

Median

$$\begin{cases}a+\sqrt{\dfrac{(b-a)(c-a)}{2}} & \text{if } c\ge\dfrac{b-a}{2}\\[2ex] b-\sqrt{\dfrac{(b-a)(b-c)}{2}} & \text{otherwise}\end{cases}$$

Mode

c

PsiUniform

```
PsiUniform (a,b,...)
```

PsiUniform (a,b) is a flat, bounded distribution with lower bound a and upper bound b. It is used to represent a random variable that is equally likely to take on any value between a lower and upper bound. A Uniform (0,1) distribution is also known as a Standard Uniform distribution, and is used to generate many other random variables. If X is a random variable with a Standard Uniform distribution, then a + (b – a)X has a Uniform (a,b) distribution, and (1 – X) has a Standard Uniform distribution.

Parameters

$$a < b$$

Range of Function Values

$$\left[a, b\right]$$

Probability Density Function

$$f\left(x\right) = \begin{cases} \dfrac{1}{b-a} \text{ if } a \le x \le b \\ 0 \text{ otherwise} \end{cases}$$

Cumulative Distribution Function

$$F\left(x\right) = \begin{cases} 0 \text{ if } x < a \\ \dfrac{x-a}{b-a} \text{ if } a \le x \le b \\ 1 \text{ if } x > b \end{cases}$$

Mean

$$\frac{a+b}{2}$$

Variance

$$\frac{\left(b-a\right)^2}{12}$$

Skewness

0

Kurtosis

9/5

Median

$$\frac{a+b}{2}$$

Mode

Any value in [a,b]

PsiWeibull

`PsiWeibull (α,β,...)`

PsiWeibull (α,β) is a distribution with a finite lower bound of 0. The Weibull distribution is quite flexible and can be used to model weather patterns, material strength, processing and delivery times, and in a variety of reliability engineering applications.

If X is a random variable with Weibull (1,β) distribution, then it also has the Exponential (β) distribution. In fact, a random variable X ~Weibull (α,β) if and only if X^{α} ~ Exponential (β^{α}). Also, if X is a random variable with Weibull (2,β) distribution, then it also has the Rayleigh (β) distribution.

Parameters

$$\alpha, \beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \alpha\beta^{-\alpha}x^{\alpha-1}e^{-\left(x/\beta\right)^{\alpha}}$$

Cumulative Distribution Function

$$F(x) = 1 - e^{-\left(x/\beta\right)^{\alpha}}$$

Mean

$$\frac{\beta}{\alpha}\Gamma\left(\frac{1}{\alpha}\right)$$

$\Gamma(x)$ is the Gamma function

Variance

$$\frac{\beta^2}{\alpha}\left[2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha}\Gamma^2\left(\frac{1}{\alpha}\right)\right]$$

Skewness

$$\frac{\frac{3}{\alpha}\Gamma\left(\frac{3}{\alpha}\right) + \frac{6}{\alpha^2}\Gamma\left(\frac{2}{\alpha}\right)\Gamma\left(\frac{1}{\alpha}\right) + \frac{2}{\alpha^3}\Gamma^3\left(\frac{1}{\alpha}\right)}{\left[\frac{2}{\alpha}\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha^2}\Gamma^2\left(\frac{1}{\alpha}\right)\right]^{3/2}}$$

Kurtosis

$$\frac{\frac{-6}{\alpha^2}\Gamma\left(\frac{1}{\alpha}\right) + \frac{24}{\alpha}\Gamma^2\left(\frac{1}{\alpha}\right)\Gamma\left(\frac{2}{\alpha}\right) - 12\Gamma^2\left(\frac{2}{\alpha}\right) - 12\Gamma\left(\frac{1}{\alpha}\right)\Gamma\left(\frac{3}{\alpha}\right) + 4\alpha\Gamma\left(\frac{4}{\alpha}\right)}{\left[2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha}\Gamma^2\left(\frac{1}{\alpha}\right)\right]^2}$$

Median

$$\beta\left(\ln\left(2\right)\right)^{1/\alpha}$$

Mode

$$\beta\left(\frac{\alpha-1}{\alpha}\right)^{1/\alpha} \text{ if } \alpha \geq 1$$

0 otherwise

Discrete Analytic Distributions

See below for a list of all supported Discrete Analytic Distributions.

PsiBernoulli

```
PsiBernoulli (p,...)
```

PsiBernoulli (p) is a discrete distribution that takes on a value of 1 with probability p, and a value of 0 with probability (1-p). A Bernoulli random variable is usually considered as an outcome of an experiment with only two possible outcomes (0 and 1); each experiment is called a ‘Bernoulli Trial’.

Parameters

$$p \in [0,1]$$

Range of Function Values

$$\{0,1\}$$

Probability Mass Function

$$p(x) = \begin{cases} 1-p & \text{if } x = 0 \\ p & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1-p & \text{if } 0 \leq x < 1 \\ 1 & \text{if x} \geq 1 \end{cases}$$

Mean

$$p$$

Variance

$$p(1-p)$$

Skewness

$$\frac{1-2p}{\sqrt{p(1-p)}}$$

Kurtosis

$$\frac{6p^2-6p+1}{p(1-p)}$$

Median

Not defined

Mode

$$\begin{cases} 0 \text{ if } p < \frac{1}{2} \\ 1 \text{ if } p > \frac{1}{2} \\ 0 \text{ and } 1 \text{ if } p = \frac{1}{2} \end{cases}$$

PsiBinomial

`PsiBinomial (n,p,...)`

PsiBinomial (n,p) is a discrete distribution of the number of successes in n independent 'Bernoulli Trials' (experiments with exactly two possible outcomes), where p is the success probability in each trial. The Binomial distribution can be used to model the number of winning trades in a trading system, or the number of defective items in a batch.

A random variable X is defined by X ~ PsiBinomial (n,p) if and only if n-X ~ PsiBinomial (n,1-p).

The Poisson distribution with parameter λ is a good approximation of the PsiBinomial (n,p) distribution when $n \to \infty$ and $p \to 0$, with $\lambda = np$.

Parameters

$$n > 0, \text{ integer}$$

$$p \in [0,1]$$

Range of Function Values

$$\{0,1,\ldots,n\}$$

Probability Mass Function

$$p(x) = \begin{cases} \binom{n}{x} p^x (1- p)^{n-x} & \text{if } x \in \{0,1,\ldots,n\} \\ 0 & \text{otherwise} \end{cases}$$

where $\binom{n}{x}$ is the binomial coefficient,

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ \sum_{i=0}^{\lfloor x \rfloor} \binom{n}{i} p^i (1- p)^{n-i} & \text{if } 0 \le x \le n \\ 1 & \text{if } x > n \end{cases}$$

Mean

$$np$$

Variance

$$np(1-p)$$

Skewness

$$\frac{1-2p}{\sqrt{np(1-p)}}$$

Kurtosis

$$\frac{6p^2 - 6p + 1}{np(1-p)}$$

Median

one of $\{\lfloor np \rfloor, \lfloor np \rfloor - 1, \lfloor np \rfloor + 1\}$

Mode

$$\begin{cases} p(n+1) \text{ and } p(n+1)-1 \text{ if } p(n+1) \text{ is integer} \\ \lfloor p(n+1) \rfloor \text{ otherwise} \end{cases}$$

PsiGeometric

```
PsiGeometric (p,...)
```

PsiGeometric (p) is a discrete distribution of the number of failures before the first success in a sequence of independent ‘Bernoulli Trials’ (experiments with exactly two possible outcomes), where p is the success probability in each trial. The Geometric distribution can be used to model the number of losing trades before the first winning trade, the number of items passing inspection before the first defective item appears in a batch, etc.

PsiGeometric (p) can be considered as a discrete analog of the (continuous) Exponential distribution. If X_1, X_2, …, X_n are independent geometrically distributed random variables with parameters p_1, p_2, …,p_n, then X = min (X_1, X_2, …, X_n) is also a Geometrically distributed random variable with parameter p = 1 – [(1-p_1)(1-p_2)…(1-p_n)]. Additionally, if X_1, X_2, …, X_n are Geometrically distributed random variables with parameter p, then their sum is Negative Binomially distributed with parameters n, p.

Parameters

$$p \in [0,1]$$

Range of Function Values

$$\{0,1,\dots\}$$

Probability Mass Function

$$p(x) = \begin{cases} p(1- p)^{x} \text{ if } x \in \{0,1,\dots\} \\ 0 \text{ otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 1-(1- p)^{\lfloor x \rfloor+1} \text{ if } x \geq 0 \\ 1 \text{ otherwise} \end{cases}$$

Mean

$$\frac{1-p}{p}$$

Variance

$$\frac{1-p}{p^2}$$

Skewness

$$\frac{2-p}{\sqrt{(1-p)}}$$

Kurtosis

$$\frac{9p^2-17p+9}{(1-p)^2}$$

Median

$$\frac{\ln(0.5)}{\ln(1-p)}-1$$

Mode

0

PsiHyperGeo

`PsiHyperGeo (n,D,M,...)`

PsiHyperGeo (n,D,M) is a discrete distribution of the number of successes in n successive trials drawn without replacement from a finite population of size M, when it is known that there are exactly D failures in the population. The Hypergeometric distribution can be used to model ‘good’ and defective parts in a manufacturing process.

A Hypergeometric distribution can be approximated by a Binomial distribution with parameters n, p = D/M, when M is very large as compared to n.

Parameters

$$M \in \{0,1,\dots\}$$

$$n, D \in \{0,1,\dots,M\}$$

Range of Function Values

$$\{\max(0, n-M+D),\dots,\min(D,n)\}$$

Probability Mass Function

$$p(x) = \frac{\binom{D}{x}\binom{M-D}{n-x}}{\binom{M}{n}}$$

Cumulative Distribution Function

$$F(x) = \sum_{i=1}^{x} \frac{\binom{D}{i}\binom{M-D}{n-i}}{\binom{M}{n}}$$

Mean

$$\frac{nD}{M}$$

Variance

$$\frac{n\left(D/M\right)(M-n)\left(1-D/M\right)}{M-1}$$

Skewness

$$\frac{(M-2D)(M-2n)}{M-2}\sqrt{\frac{(M-1)}{nD(M-D)(M-n)}}$$

Kurtosis

$$\left[\frac{M^2(M-1)}{n(M-2)(M-3)(M-n)}\right]\left[\frac{M(M+1)-6M(M-n)}{D(M-D)}+\frac{3n(M-n)(N+6)}{M^2}-6\right]$$

Median

Not defined

Mode

$\frac{(n+1)(D+1)}{M+2}$ and $\frac{(n+1)(D+1)}{M+2}-1$ if $\frac{(n+1)(D+1)}{M+2}$ is integral

$\left\lfloor\frac{(n+1)(D+1)}{M+2}\right\rfloor$ otherwise

PsiIntUniform

```
PsiIntUniform (a,b,...)
```

PsiIntUniform (a,b) is a discrete distribution with equal probability at each integer value between the lower and upper bounds (a and b). It is used as a rough estimate of the true distribution when the only information we have is that the random variable takes integer values between a and b, and each of these values are equally likely.

Parameters

a, b integers

a<b

Range of Function Values

$$\{a, a+1, \ldots, b-1, b\}$$

Probability Mass Function

$$p(x)=\begin{cases}\dfrac{1}{b-a+1}\text{ if } a\le x\le b, \text{ and } x \text{ integer}\\ 0 \text{ otherwise}\end{cases}$$

Cumulative Distribution Function

$$F(x)=\begin{cases}0 \text{ if } x<a\\ \dfrac{\lfloor x\rfloor-a+1}{b-a+1}\text{ if } a\le x\le b\\ 1 \text{ if } x>b\end{cases}$$

Mean

$$\frac{a+b}{2}$$

Variance

$$\frac{(b-a+1)^2-1}{12}$$

Skewness

0

Kurtosis

$$-\frac{6\left\{(b-a+1)^2+1\right\}}{5\left\{(b-a+1)^2-1\right\}}$$

Median

$$\frac{a+b}{2}$$

Mode

Net defined

PsiLogarithmic

```
PsiLogarithmic (p,...)
```

PsiLogarithmic is a discrete distribution with a lower bound of 1. It is used to describe the diversity of a sample.

Parameters

$$p \in (0,1)$$

Range of Function Values

$$\{1,2,3,\ldots\}$$

Probability Mass Function

$$p(x)=\begin{cases}\dfrac{-1}{\ln(1-p)}\dfrac{p^x}{x} & \text{if } x \geq 1 \\ 0 \text{ otherwise}\end{cases}$$

Cumulative Distribution Function

$$F\left(x\right)=1+\frac{B_p\left(x+1,0\right)}{\ln\left(1-p\right)},\ B_p\left(a,b\right) \text{ is the incomplete beta function}$$

$$B_p\left(a,b\right)=\int_0^p t^{a-1}\left(1-t\right)^{b-1}dt$$

Mean

$$\frac{-p}{\left(1-p\right)\ln\left(1-p\right)}$$

Variance

$$-p\frac{p+\ln\left(1-p\right)}{\left(1-p\right)^2\ln^2\left(1-p\right)}$$

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

1
PsiNegBinomial

```
PsiNegBinomial (s,p,...)
```

PsiNegBinomial (s,p) is a discrete distribution that describes the number of failures that will occur before a given number of successes, where each trial is successful with probability p. The Negative Binomial distribution can be used to describe the number of items that pass inspection before the sth defective item is found.

If X_1, X_2,…,X_s are independent Geometrically distributed random variables each with parameter p, then their sum is Negative Binomially distributed, with parameters s and p. Additionally, the Geometric distribution with parameter p is the same as a Negative Binomial distribution with parameters s = 1 and p; hence, the Geometric distribution is a special case of a Negative Binomial distribution.

Parameters

$$p\in\left[0,1\right]$$

$$s>0,\ \text{integer}$$

Range of Function Values

$$\left\{0,1,2,3,\ldots\right\}$$

Probability Mass Function

$$p(x)=\begin{cases}\binom{s+x-1}{x}p^s(1-p)^x \text{ if } x\in\{0,1,\ldots\}\\ 0 \text{ otherwise}\end{cases}$$

Cumulative Distribution Function

$$F(x)=\begin{cases}\sum_{i=0}^{\lfloor x\rfloor}\binom{s-i+1}{i}p^s(1-p)^i \text{ if } x\geq 0\\ 0 \text{ otherwise}\end{cases}$$

Mean

$$\frac{s(1-p)}{p}$$

Variance

$$\frac{s(1-p)}{p^2}$$

Skewness

$$\frac{2-p}{\sqrt{s(1-p)}}$$

Kurtosis

$$\frac{p^2-6p+1}{s(1-p)}+3$$

Median

Not defined

Mode

$$\begin{cases}\frac{s(1-p)-1}{p} \text{ and } \frac{s(1-p)-1}{p}+1 \text{ if } \left(\frac{s(1-p)-1}{p}\right) \text{ is integer}\\ \left\lfloor\frac{s(1-p)-1}{p}\right\rfloor+1 \text{ otherwise}\end{cases}$$

PsiPoisson

```
PsiPoisson (λ,...)
```

PsiPoisson (λ) is a discrete distribution of the number of events that occur in an interval of time, when the events occur at a known average rate, and each occurrence is independent of the time of occurrence of the previous event.

The Poisson distribution with parameter λ can be approximated by a Normal distribution with mean λ and variance λ, for large values of λ. If X_1, X_2,…,X_n are independent Poisson random variables with parameters λ_1, λ_2,… λ_n, then their sum is also a Poisson random variable with parameter $\lambda_1 + \lambda_2 + \ldots + \lambda_n$.

Parameters

$\lambda > 0$

Range of Function Values

$$\{0,1,\ldots\}$$

Probability Mass Function

$$p(x) = \begin{cases} \dfrac{e^{-\lambda}\lambda^x}{x!} \text{ if } x \in \{0,1,\ldots\} \\ 0 \text{ otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 \text{ if } x < 0 \\ e^{-\lambda}\sum_{i=0}^{\lfloor x \rfloor} \dfrac{\lambda^{\mathrm{i}}}{i!} \text{ if } x \ge 0 \end{cases}$$

Mean

λ

Variance

λ

Skewness

$$\sqrt{\frac{1}{\lambda}}$$

Kurtosis

$$\frac{1}{\lambda} + 3$$

Median

Not applicable

Mode

$$\begin{cases} \lambda \text{ and } \lambda - 1 \text{ if } \lambda \text{ is integer} \\ \lfloor \lambda \rfloor \text{ otherwise} \end{cases}$$

Custom Distributions

PsiCumul

`PsiCumul (a,b, {x1,x2,…,xn}, {p1,p2,…,pn},...)`

PsiCumul (a,b, $\{x_1,x_2,\ldots,x_n\},\{p_1,p_2,\ldots,p_n\},\ldots$) is a custom continuous distribution with lower and upper bounds equal to a and b respectively, and with user specified values , $x_1,x_2,\ldots,x_n$ and corresponding cumulative probabilities $p_1,p_2,\ldots,p_n$.

Parameters

$$a < b$$

$$a \le x_i \le b \forall i = 1,2,\ldots,n$$

$$0 \le p_i \le 1 \forall i = 1,2,\ldots,n$$

$$p_i < p_{i+1} \forall i = 1,2,\ldots,n-1$$

$$x_i < x_{i+1} \forall i = 1,2,\ldots,n-1$$

Define the boundary parameters as

$$x_0 = a, x_{n+1} = b, p_0 = 0, p_{n+1} = 1$$

Range of Function Values

$$\left[a,b\right]$$

Probability Density Function

$$f\left(x\right) = \frac{p_{i+1} - p_i}{x_{i+1} - x_i} \text{ if } x_i \le x \le x_{i+1}$$

Cumulative Distribution Function

$$F\left(x\right) = p_i + \left(p_{i+1} - p_i\right)\left(\frac{x - x_i}{x_{i+1} - x_i}\right) \text{ if } x_i \le x \le x_{i+1}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiDiscrete

`PsiDiscrete ({x1,x2,…,xn}, {p1,p2,…,pn},…)`

PsiDiscrete ($\{x_1,x_2,\ldots,x_n\}$, $\{p_1,p_2,\ldots,p_n\}$,…) is a custom discrete distribution that takes on values $\{x_1,x_2,\ldots,x_n\}$ with probabilities $\{p_1,p_2,\ldots,p_n\}$ respectively.

Parameters

$$\{x_1, x_2, \ldots, x_n\}$$

$$\{p_1, p_2, \ldots, p_n\}$$

The probabilities p_i are first normalized so that they sum to one

Range of Function Values

$$\{x_1, x_2, \ldots, x_n\}$$

Probability Density Function

$$f(x) = \begin{cases} p_i \text{ if } x = x_i \\ 0 \text{ otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 \text{ if } x < x_1 \\ \sum_{i=1}^{s} p_i \text{ if } x_s \le x < x_{s+1}, s < n \\ 1 \text{ otherwise} \end{cases}$$

This assumes that $x_i \le x_{i+1} \forall i = 1, 2, \ldots, n-1$

Mean

$$\sum_{i=1}^{n} x_i p_i = \mu$$

Variance

$$\sum_{i=1}^{n} (\mu - x_i)^2 p_i = \sigma^2$$

Skewness

$$\frac{\sum_{i=1}^{n} (x_i - \mu)^3 p_i}{\sigma^3}$$

Kurtosis

$$\frac{\sum_{i=1}^{n} (x_i - \mu)^4 p_i}{\sigma^4}$$

Median

$$x_s \text{ where } s = \min\left(j = 1, 2, \ldots, n : \sum_{i=1}^{j} p_i \geq 0.5 \right)$$

This assumes that $x_i \leq x_{i+1} \forall i = 1, 2, \ldots, n-1$

Mode

$$x_{\underset{i=1,2,\ldots,n}{\arg\max}(p_i)}$$

PsiDisUniform

`PsiDisUniform ({x`$_1$`,x`$_2$`,…,x`$_n$`},…)`

PsiDisUniform ({x_1,x_2,…,x_n},…) is a custom discrete distribution that takes on values {x_1,x_2,…,x_n} with equal probability. It is similar to the PsiDiscrete distribution except that no probabilities are specified – instead all x values are equally likely to occur. (In the equations below, each p_i = 1/n.) PsiDisUniform can be used to *resample* a set of past observations {x_1,x_2,…,x_n}.

Parameters

$$\{x_1, x_2, \ldots, x_n\}$$

These values have the corresponding probabilities as

$$p_i = \frac{1}{n} \forall i = 1, 2, \ldots, n$$

Range of Function Values

$$\{x_1, x_2, \ldots, x_n\}$$

Probability Density Function

$$f(x) = \begin{cases} p_i \text{ if } x = x_i \\ 0 \text{ otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 \text{ if } x < x_1 \\ \sum_{i=1}^{s} p_i \text{ if } x_s \leq x < x_{s+1}, s < n \\ 1 \text{ otherwise} \end{cases}$$

This assumes that $x_i \leq x_{i+1} \forall i = 1, 2, \ldots, n-1$

Mean

$$\sum_{i=1}^{n} x_i p_i = \mu$$

Variance

$$\sum_{i=1}^{n}(\mu - x_i)^2 p_i = \sigma^2$$

Skewness

$$\frac{\sum_{i=1}^{n}(x_i - \mu)^3 p_i}{\sigma^3}$$

Kurtosis

$$\frac{\sum_{i=1}^{n}(x_i - \mu)^4 p_i}{\sigma^4}$$

Median

$$x_s \text{ where } s = \min\left(j = 1, 2, \ldots, n : \sum_{i=1}^{j} p_i \ge 0.5 \right)$$

$$\text{This assumes that } x_i \le x_{i+1} \forall i = 1, 2, \ldots, n-1$$

Mode

$$x_{\underset{i=1,2,\ldots,n}{\arg\max}(p_i)}$$

PsiGeneral

`PsiGeneral (a,b, {x1,x2,…,xn}, {w1,w2,…,wn},…)`

PsiGeneral (a,b, $\{x_1,x_2,\ldots,x_n\}$, $\{w_1,w_2,\ldots,w_n\}$,…) is a custom continuous distribution with lower and upper bounds equal to a and b respectively, and with user specified values $x_1,x_2,\ldots,x_n$ and corresponding weights $w_1,w_2,\ldots,w_n$. This is similar to a PsiCumul (a,b, $\{x_1,x_2,\ldots,x_n\}$,$\{p_1,p_2,\ldots,p_n\}$,…) distribution, where the probabilities are calculated using the weights as shown below.

Parameters

$$a < b$$

$$a \le x_i \le b \forall i = 1, 2, \ldots, n$$

$$x_i < x_{i+1} \forall i = 1, 2, \ldots, n-1$$

$$\text{The cumulative probabilities are defined as } p_i = \sum_{k=1}^{i} \frac{w_k}{\sum_{j=1}^{n} w_j}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f\left(x\right)=\frac{p_{i+1}-p_i}{x_{i+1}-x_i}\text{ if } x_i \le x \le x_{i+1}$$

Cumulative Distribution Function

$$F\left(x\right)=p_i+\left(p_{i+1}-p_i\right)\left(\frac{x-x_i}{x_{i+1}-x_i}\right)\text{ if } x_i \le x \le x_{i+1}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiHistogram

`PsiHistogram (a,b,{w1,w2,…,wn},…)`

PsiHistogram (a,b,{w_1,w_2,…,w_n},…) is a custom continuous distribution with lower and upper bounds equal to a and b respectively, and with user specified weights w_1,w_2,…,w_n corresponding to n subintervals of equal size. This is similar to a PsiCumul (a,b, {x_1,x_2,…,x_n},{p_1,p_2,…,p_n},…) distribution, where the probabilities are calculated using the weights as shown below, and the interval defined by the bounds a and b is divided into subintervals of equal size as described below.

Parameters

$a<b$

The interval $\left[a,b\right]$ is divided into n subintervals

of equal size $\left\{x_1,x_2,\ldots,x_n\right\}$

$$x_i=a+i\left(\frac{b-a}{n}\right)$$

The cumulative probabilities are defined as $p_i=\sum_{k=1}^{i}\frac{w_k}{\sum_{j=1}^{n}w_j}$

Range of Function Values

$$[a,b]$$

Probability Density Function

$$f(x)=\frac{p_{i+1}-p_i}{x_{i+1}-x_i} \text{ if } x_i \le x \le x_{i+1}$$

Cumulative Distribution Function

$$F(x)=p_i+(p_{i+1}-p_i)\left(\frac{x-x_i}{x_{i+1}-x_i}\right) \text{ if } x_i \le x \le x_{i+1}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

Special Distributions

The RASON modeling language offers a number of special PSI Distribution functions that do not fit readily into the classes of continuous, discrete and custom distributions described above. For example, PsiSip() and PsiSlurp() ensure that Monte Carlo trials are drawn *sequentially* from SIP or SLURP data. And PsiMVNormal(), PsiMVLogNormal(), PsiResample() and PsiShuffle() return array results rather than single-valued results.

Note: **PSI Property functions** generally may **not** be passed as arguments to any of the PSI Distribution functions in this section. The only exception is that the PsiCertify() function may be passed to PsiSip() or PsiSlurp(), enabling the SIP or SLURP-based distribution to be named and published as a Certified Distribution.

Functions PsiMVLogNormal(), PsiMVNormal(), PsiResample() and PsiShuffle() are included to provide an upgrade path for users of AnalyCorp’s XLSim software. Note that PsiMVLogNormal() and PsiMVNormal() require a *covariance* matrix (not a rank correlation matrix) as an argument; they cannot be correlated with dissimilar distributions specified via other PSI Distribution functions.

PsiFit

```
PsiFit (data)
```

PsiFit dynamically fits a probability distribution to sample data, and creates an uncertain variable linked to the sample data. This dynamically fitted uncertain variable can then be used in the model as an uncertain input variable. The "data" argument is a list of sample data. The examples below illustrates how to use PsiFit in two different ways: PsiFit({list, of , values}) or PsiFit(cell_range).

Example 1:

```
"uncertainFunctions": {
     "testFit": {
            "formula": "PsiFit({1,2,3,4,5,6,7,8,9,10})",
            "mean": [],
            "percentiles": [],
            "trials": []
     }
}
```

Example 2:

```
"data": {
      "A1:A10": {
            "value": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        }
    },
"uncertainFunctions": {
     "testFit": {
            "formula": "PsiFit(A1:A10)",
            "mean": [],
            "percentiles": [],
            "trials": []
     }
}
```

PsiMVLogNormal

`PsiMVLogNormal (µ,∑)`

PsiMVLogNormal (µ,∑) is a *multivariate* distribution that returns a vector of random variables that are lognormally distributed, with mean values specified by the vector µ, and covariance values specified by the matrix ∑. This is a generalization of the PsiLogNorm2 distribution to higher dimensions. A variable vector y = [y1,…,yn] has a multivariate LogNormal distribution if and only if the variable vector [ln(y1),…,ln(yn)] has a multivariate Normal distribution.

PsiMVLogNormal returns an *array* of sample data**.** On each Monte Carlo trial, the function will return 5 sample values.

Parameters

μ, a real vector

A positive semidefinite matrix $\sum$

We define parameters $\sigma_i^2 = \sum_{ii}$

Range of Function Values

$\left[0,\infty\right)^n$ for an *n*-dimensional vector

Probability Density Function

$$f\left(y\right)=\left(2\pi\right)^{-n/2}\left(y\right)^{-1}\left|\Sigma\right|^{-1/2}\exp\left[\frac{-\left(\ln\left(y\right)-\mu\right)^{T}\Sigma^{-1}\left(\ln\left(y\right)-\mu\right)}{2}\right]$$

Cumulative Distribution Function

No closed form

Mean

$$\left[e^{\mu_1+\sigma_1^2/2},\ldots,e^{\mu_n+\sigma_n^2/2}\right]$$

Variance

$$\left[\left(e^{\sigma_1^2}-1\right)e^{2\mu_1+\sigma_1^2},\ldots,\left(e^{\sigma_n^2}-1\right)e^{2\mu_n+\sigma_n^2}\right]$$

Skewness

$$[s_1,\ldots,s_n]$$

where

$$\mathrm{s_i}=\left(e^{\sigma_i^2}+2\right)\sqrt{e^{\sigma_i^2}-1}$$

Kurtosis

$$[k_1,\ldots,k_n]$$

where

$$k_i=e^{4\sigma_i^2}+2e^{3\sigma_i^2}+3e^{2\sigma_i^2}-3$$

Median

$$\left[e^{\mu_1},\ldots,e^{\mu_n}\right]$$

Mode

$$\left[e^{\mu_1-\sigma_1^2},\ldots,e^{\mu_n-\sigma_n^2}\right]$$

PsiMVNormal

```
PsiMVNormal (µ,∑)
```

PsiMVNormal (µ,∑) is a *multivariate* distribution that returns a vector of random variables that are normally distributed, with mean values specified by the vector µ, and covariance values specified by the matrix ∑. This is a generalization of the PsiNormal distribution to higher dimensions.

PsiMVNormal returns an *array* of sample data. On each Monte Carlo trial, the function will return 5 sample values.

Parameters

$\mu,$ a real vector

A positive definite matrix Σ

We define parameters $\sigma_i^2 = \Sigma_{ii}$

Range of Function Values

$(-\infty,\infty)^n$ for an n-dimensional vector

Probability Density Function

$$f(y) = (2\pi)^{-n/2} |\Sigma|^{-1/2} \exp\left[\frac{-(y-\mu)^T \Sigma^{-1} (y-\mu)}{2} \right]$$

Cumulative Distribution Function

No closed form

Mean

$[\mu_1, \ldots, \mu_n]$

Variance

$[\sigma_1^2, \ldots, \sigma_n^2]$

Skewness

$[0]^n$ for an n-dimensional vector

Kurtosis

$[0]^n$ for an n-dimensional vector

Median

$[\mu_1, \ldots, \mu_n]$

Mode

$[\mu_1, \ldots, \mu_n]$

PsiResample

```
formula: "PsiResample(data)"
```

PsiResample returns an *array* of sample data. On each Monte Carlo trial, the function will return 5 sample values.

PsiResample returns a random sample (with replacement) of the trial values in array specified by the *data* argument.

PsiMVResample

`formula: "PsiMVResample(data)"`

PsiMVResample is a *multivariate* distribution that returns a vector of sample data. PsiMVResample returns a random sample of the trial values specified by the *data* argument. On each Monte Carlo trial, PsiMVResample will return a uniformly selected column from the argument.

PsiShuffle

`formula: "PsiShuffle(data)"`

PsiShuffle returns an *array* of sample data. On each Monte Carlo trial, the function will return 10 sample values. PsiShuffle returns a random permutation of all the trial values specified by the *data* argument. In the sample drawn on each Monte Carlo trial, each value in the *data* range is selected only once; values are repeated in a single sample only if they are repeated in *data*.

PsiMVShuffle

`formula:"PsiMVShuffle(data)"`

PsiMVShuffle is a *multivariate* distribution that returns a vector of sample data. On each Monte Carlo trial, the function will return 10 sample vectors. PsiMVShuffle returns a random permutation of all the trial values in the cell range specified by the *data* argument. In the sample drawn on each Monte Carlo trial, each vector in the *data* range is selected only once; values are repeated in a single sample only if they are repeated in *data*.

PsiSip

`formula:"PsiSip(sip)"`

PsiSip returns trials for an uncertain variable from a list or vector of sample data, called a Stochastic Information Packet (SIP). The *sip* argument is an Excel cell range containing the list of sample data. The value returned by PsiSip() on the *i*th trial is the *i*th value in the list.

PsiSlurp

`formula: "PsiSlurp(slurp,j)"`

PsiSlurp returns trials for an uncertain variable from a table of correlated sample data, called a Stochastic Library Unit, Relationships Preserved (SLURP), in the sequence specified in the table. The *slurp* argument is an array containing the SLURP data; the *j* argument is the index of the desired SIP (column) of the SLURP, starting from 1. The value returned by PsiSlurp() on the *i*th trial is taken from the *i*th row and the *j*th column of the table.

PSI Property Functions

Using PSI Property Functions

PSI Property functions should be entered *only* as additional arguments of analytic and custom PSI Distribution functions. They modify the behavior of the PSI Distribution function in which they appear.

For example, `formula: "PsiNormal (0, 1)"` specifies a Normal distribution with mean 0 and standard deviation 1: Sample values drawn from this distribution *could* be any number from ‘minus infinity’ to ‘plus infinity’ (though sample values near 0 are more *likely* to be drawn). If you write `formula: "PsiNormal`

`(0, 1, PsiTruncate (-10, 10))"` the distribution is ‘truncated’ so that sample values *always* lie within the range from -10 to +10.

You can specify more than one PSI Property function as an argument to a PSI Distribution function, and they can appear in any order after the required arguments. For example, `formula: "PsiBeta (1, 2, PsiTruncate (-10, 10), PsiShift(3), PsiCorrDepen("MyCorr", 0.5))"` specifies a Beta distribution with shape parameters 1 and 2, truncated to a range from -10 to +10, shifted right by 3, and correlated with the uncertain variable whose definition contains PsiCorrIndep ("MyCorr"), with rank correlation coefficient 0.5.

PsiBaseCase

`formula: "PsiBaseCase(value)"`

Use this property to specify a Base Case value for an Uncertain Variable. This is the single value that you’d want the uncertain variable to be if you were *not* an Uncertain Variable (i.e. if it did not have a PSI Distribution function).

PsiCertify

`PsiCertify (name, default_value, short_description, full_description, version, author, copyright, trademark, history)`

PsiCertify is used to name, certify and ‘publish’ a PSI Distribution function as a Certified Distribution. The *default_value* argument should be a number; all other arguments should be character strings. Only the *name* argument is required; the others are optional.

PsiCensor

`PsiCensor(min, max)`

PsiCensor is used to pile the values of samples from the uncertain variable’s distribution as follows: if the uncertain variable’s value is less than the Min value, then the sample value will be piled at the Min, if the uncertain variable’s value is larger than the Max value, then the sample value will be piled at the Max. This argument results in a “build up” of values around the Min and Max values in the distribution.

PsiCorrMatrix

`PsiCorrMatrix (matrix array, position, instance)`

PsiCorrMatrix is used to specify that a uncertain variable is correlated with a group of other uncertain variables, through a matrix of rank-order correlation coefficients. The first argument, *matrix array*, is a 2-dimensional array containing the correlation matrix. *Position* specifies the uncertain variable index in the correlation matrix. *Instance* is the string name given to the correlation matrix.

You pass "PsiCorrMatrix (*matrix cell range*, *position*)" as an argument to the formula in the uncertain variable PSI Distribution function, for example:

```
data: {
   corrmatrix: {
      dimensions: [3,3], value: [[1, 0.8, 0.5],[0.8, 1, 0.2],[0.5, 0.2,
      1]]
   }
 },
uncertainVariables: {
   uncVar1: {
       formula: "=PsiUniform(0,100,PsiCorrMatrix(corrmatrix, 1))",
       mean: []
     },
   uncVar2: {
          formula: "=PsiNormal(10,5,PsiCorrMatrix(corrmatrix, 2))",
          mean: []
      },
      uncVar3: {
          formula: "=PsiNormal(10, 2,PsiCorrMatrix(corrmatrix, 3))",
          mean: []
      }
 },
```

Note that `corrmatrix` is a 2-dimensional array. PsiCorrMatrix() within the uncertain variable formulas specify that the first variable has a rank correlation coefficient of 0.8 with the second variable, and 0.5 with the third variable. The second and third variables are correlated with each other, with a rank correlation coefficient of 0.2. Note that a correlation matrix must always have 1's on the diagonal, because an uncertain variable is always perfectly correlated with itself. Also, the matrix must be symmetric: If row 2, column 1 contains 0.8, then row 1, column 2 must also contain 0.8. Finally, the correlation coefficients must be consistent with each other: For example, if uncertain variable 1 is strongly positively correlated with variable 2, and variable 2 is strongly positively correlated with variable 3, then variable 1 cannot be negatively correlated with variable 3. Formally, the matrix must be *positive semidefinite* – it cannot have any negative eigenvalues.

PsiCorrDepen / PsiCorrIndep

PsiCorrDepen

`PsiCorrDepen(corrname,coefficient)`

PsiCorrDepen is used to specify that this uncertain variable is correlated with one other uncertain variable, with the specified rank-order correlation coefficient. The *corrname* argument is a text string that must match the *corrname* argument of the 'independent variable,' a cell containing a PSI distribution with the PsiCorrIndep() property function call. See below for an example.

PsiCorrIndep

`PsiCorrIndep(corrname)`

PsiCorrIndep is used to specify that this uncertain variable acts as an independent variable correlated with one other uncertain variable, the dependent variable. The *corrname* argument is a text string that must match the *corrname* argument of the related PsiCorrDepen() call.

Example

Note the use of single quotes around the `MyCoor` argument inside of the PsiCorrIndep()/PsiCorrDepen() functions.

```
uncertainVariables: {

      uncVar1: {

            formula: "=PsiUniform(0,100,PsiCorrDepen('MyCorr',0.9))",

            mean: []

      },

      uncVar2: {

            formula: "=PsiNormal(100,10,PsiCorrIndep('MyCorr'))",

            mean: []

      }

    },
```

PsiLock

`PsiLock(value)`

PsiLock is used to (temporarily) make an uncertain variable “constant,” so it returns the specified value for all trials in a simulation, regardless of the distribution function used.

PsiSeed

`PsiSeed(value)`

PsiSeed is used to set a random number seed for Monte Carlo samples generated for this distribution function, that will override any general seed value specified for the simulation model. It is most often used in an analytic distribution that is being published as a Certified Distribution.

PsiShift

`PsiShift (shift)`

PsiShift is used to shift the domain of this uncertain variable’s distribution by the specified amount.

PsiTruncate

`PsiTruncate (min,max)`

PsiTruncate is used to restrict the values of samples from this uncertain variable’s distribution to lie within the range from min to max.

Psi Data Mining/Forecasting Function Signatures

Signatures for PsiForecast(), PsiPredict(), PsiTransform() and PsiPosteriors() when utilizing a PMML model in RASON. See the Decisions Table chapter for a complete walk though of how to use one of these four functions to load a data mining or forecasting model, saved in PMML format, into RASON.

PsiForecast()

`PsiForecast(Model, Input_Data, [Simulate], [Num_forecasts], [Header] )`

Computes the forecasts for `Input_Data` using a Time Series model stored in PMML format.

`Model`: Range containing the stored Times Series model in PMML format. Note: Argument will always start with cell B12 and end with the cell address containing </PMML>.

`Input_Data`: Range containing the new Time Series data for computing the forecasts. Range must contain a header with the time series name and a sufficient number of records for the forecasting with a given model.

`Simulate`: If True, the forecasts are adjusted with random normally distributed errors. If False or omitted, the forecasts will be deterministic.

`Num_forecasts`: If True, the forecasts are adjusted with radom normally distributed errors. If False or omitted, the forecasts will be deterministic.

`Header`: If True, the forecasts are adjusted with random normally distributed errors. If False or omitted, the forecasts will be deterministic.

Output*:* A single column containing the header and forecasts for input time series. The number of produced forecasts is determined by the number of selected cells in the array-formula entry.

Supported Models:

- Arima
- Exponential Smoothing
- Double Exponential Smoothing
- Holt Winters Smoothing

PsiPredict()

```
PsiPredict(Model, Input_Data, [Header])
```

Predicts the response, target, output or dependent variable for `Input_Data` whether it is continuous (Regression) or categorical (Classification) when the model is stored in PMML format. In addition, this function also computes the fitted values for a Time Series model when the model is stored in PMML format.

`Model`: Range containing the stored Classification, Regression or TimeSeries model in PMML format.

`Input_Data`: Range containing the new data for computing predictions. Range must contain a header row with column names and at least one row of data containing the exact same features (or columns) as the data used to create the model.

`Header`: If True, the forecasts are adjusted with random normally distributed errors. If False or omitted, the forecasts will be deterministic.

Output*:* A vector array containing the header and predicted/fitted values for each record in `Input_Data`.

To know if the result of the prediction is continuous or categorical, you must know what kind of model you are passing as an argument to the scoring function – if you previously fitted the classification model and are now predicting the new feature vectors, you should expect to get the compatible categorical response. On the other hand, you should expect the continuous response from the new data prediction when using a fitted regression model. Note: If the user intends to use an "unknown" model for scoring, the stored worksheets contain the complete information about the model including several clear indications of the model type and data dictionaries with the types of features and response.

PsiPredict() can compute the *fitted values* for the new time series based on the provided Time Series model. Unlike future-looking forecasting, provided by PsiForecast(), PsiPredict() computes a model *prediction* for each observation in the provided new time series.

Supported Models:

- Classification:
  - Discriminant Analysis
  - Logistic Regression
  - K-Nearest Neighbors
  - Classification Tree

  - Naïve Bayes
  - Neural Network
  - Random Trees
  - Bagging (with any supported weak learner)
  - Boosting (with any supported weak learner)
- Regression:
  - Logistic Regression
  - K-Nearest Neighbors
  - Neural Network
  - Bagging (with any supported weak learner)
  - Boosting (with any supported weak learner)
- Time Series (fitted values)
  - ARIMA
  - Exponential Smoothing
  - Double Exponential Smoothing
  - Holt-Winters Smoothing

PsiPosteriors()

`PsiPosteriors(Model, Input_Data, [Header])`

Computes the posterior probabilities for `Input_Data` using a Classification model stored in PMML format.

`Model`: Range containing the stored Classification model in PMML format. Note: Argument will always start with cell B12 and end with the cell address containing </PMML>.

`Input_Data`: Range containing the new data for computing posterior probabilities. Range must contain a header with column names and at least one row of data containing the exact same features (or columns) as the data used to create the model.

`Header`: If True, the forecasts are adjusted with random normally distributed errors. If False or omitted, the forecasts will be deterministic.

Output*:* Multiple columns containing a header with class labels and estimated posterior probabilities for each class label for all records in `Input_Data`.

Supported Models:

- Classification:
  - Discriminant Analysis
  - Logistic Regression
  - K-Nearest Neighbors
  - Classification Tree
  - Naïve Bayes
  - Neural Network
  - Random Trees
  - Bagging (with any supported weak learner)
  - Boosting (with any supported weak learner)

PsiTransform()

`PsiTransform(Model, Input_Data, [Header])`

Transforms the `Input_Data` using a Transformation model stored in PMML format.

`Model`: Range containing the stored Transformation model in PMML format. Note: Argument will always start with cell B12 and end with the cell address containing </PMML>.

`Input_Data`: Range containing the new data for transformation. Range must contain a header with column names and at least one row of data containing the exact same features (or columns) as the data used to create the model.

Header: If True, the forecasts are adjusted with random normally distributed errors. If False or omitted, the forecasts will be deterministic.

Output*:* One or multiple columns containing a header and transformed data.

Supported Models:

- Transformation:
    - Rescaling
- Text Mining
    - TF-IDF Vectorization (input data – text variable with the corpus of documents)
    - LSA Concept Extraction (input data – term-document matrix, where columns represent terms and rows represent documents)

Appendix II RASON Error Codes

Introduction

This chapter documents the RASON Error messages that can be returned when you optimize a model, run a simulation or perform a data mining function

Error Messages

General JSON Error	Indicates that an unexpected internal error has occurred.
Missing model file or string	Indicates that an internal error has occurred by can appear when a user attempts to solve an empty model string.
Model type (Simulation, optimization, data mining) mismatch	Indicates that the User has used the wrong RASON end point to solve the current model. For example, clicking Simulate in the RASON IDE rather than Solve when solving an optimization model, or calling `Get/POST rason.net/api/model/id/optimize` when running a simulation model.
Improper engine selected for the particular model	Indicates that an appropriate engine has not been selected to solve the model.
Invalid Json token	Not in use in RASON 2.0 and later versions.
Unrecognized Json identifier	Indicates that a name of a section or a property has been used which is not part of the RASON syntax. For example, if a user misspells a section heading such as “modelSetings” or a non-existent property name is passed.
Invalid Json data type	Indicates that the user has entered a different type than expected. For example, if a number is required, but a string is passed, or if an array is required, but a scaler has been passed. This error message may also be returned when an Excel type not presented in RASON is attempted in conversion (for example, Excel errors #N/A, #NUM, etc.).

Missing name definition in Json object	All objects must be named. If the “name” property is missing, this error will appear.
	When using the syntax:
	Variables:{
	x:{value:0}
	}
	the variable is implicitly named “x”.

	However, when using the syntax below, you must specify the name of the object by using the name property. Variables:[{ name: “x”, value: 0}]
Expecting ':'	Missing “:”
Expecting '{' or '['	Missing an expected opening bracket
Expecting '}' or ']'	Missing an expected closing bracket
Incompatible Json assignment	In RASON models containing a property or object with the ‘:’ operator, examples may include, the dimensions of the right hand side of an optimization constraint not matching the dimensions of the left hand side of the constraint.
Duplicated Json assignment	Indicates that an object has been defined twice.
The identifier, to which a value is assigned, must be a scalar.	The variable/parameter should be a scalr (not arry). Setting "value:[]" is not supported.
Incorrect Json array dimensions	Indicates that the definition of dimensions: [r,c] is not correct. Note: Although [r][c] is correct C++ syntax, it is not correct RASON syntax.
Incorrect Json data array	Indicates a mistake in any array definition through the [] operator, for example, [1, 2]].
Mismatching sizes of JSON arrays	For example: value:[] has different dimensions then lower: [] or upper: []
Less elements assigned to a JSON array	If a variable is defined explicitly as array through dimensions: [] and less elements were assigned through value: [].
More elements assigned to a JSON array	If a variable is defined explicitly as array through dimensions: [] and more elements were assigned through value: [].
Incorrect assignment to a Json ‘type’ identifier	Indicates that “type:” property has been assigned an invalid value. For example, “type: “bynari”” rather than “type: binary”” when applied to a decision variable or passing “type: “maximum”” to a variable or constraint instead of the objective function.
Missing Json Variable Definition	Indicates that in a conic constraint definition, an identifier has been used, which is not a decision variable.
Index [] misused or out of range	Indicates that the application of the range operator [] to an array is invalid. For example, x[6] while x has length 5.
Wrong Simulation Index in modelSettings	When running a specific simulation, out of multiple simulations, but simulationIndex property is set to <1 or > numSimulations.
Incompatible model block definition	Could indicate that a two sided constraint has been passed. For example, a constraint block with a lower bound of 1 and an upper bound of 10.

Invalid binding definition or incompatible dimensions	Indicates an error in a data source definition or the related index sets. If using PsiDataSrc() fumction in an Excel model, confirm that there are no trailing or leading spaces in any of the argument definitions in either the excel cells or the function definition.
Missing binding value column	valueCol: "" property is required when binding to data
Formulas not allowed in this definition	Indicates that a formula has been entered for an object that should not have a formula, for example, a decision variable object.
Invalid parameter definition	Indicates that PsiOptParam() or PsiSimParam() has been misspelled or entered improperly
Inconsistent Table	Indicates that an invalid table definition has been used in the functions SELECT or PIVOT; the index operator [] has been applied to the identifier, which is an invalid table; or something is wrong in the table definition either inline (in the table) or through binding.
Inconsistent Data source definition	Indicates an error in the PsiDataSrc() function in Excel during the RASON conversion. If not using Excel, then this error indicates that a data source definition and /or the datasets associated with the data source definition are invalid.
Empty data-source	The data source is empty.
Do not mix {indexCols, valueCols} and {colIndex, rowIndex} indexing systems	Two sets may not be mixed when describing a table. For example, indexCols and rowIndex where indexCols creates a table like data-source while indexCols creates a dataframe.
direction: 'export' is required in a datasource definition for saving	If exporting results, such as final variable values, the data-source must contain the property "direction": "export" in order to prevent overwriting of a data-source used only for reading.
Inconsistent arguments to SELECT function	Indicates that the is an error in the syntax for the SELECT function.
Inconsistent arguments to PIVOT function	Indicates that the is an error in the syntax for the PIVOT function.
Misused equal/valueof property	If the “equal:” property is used outside of the Constraints section.
Inconsistent index set, index column, or mismatching index set and column	Indicates an invalid index set definition or the usage of an index set in a table.
Index column values does not match the designated index set. Try the datasource property sortIndexCols:true"	One and the same element reappears in an index set; or the index column does not match a predefined index set, in the indexSets section.
Incorrect loop definition	Indicates a syntax mistake in a loop/for definition.
Array bounds not allowed in a table assignment	Not in use in RASON 2.0 and later versions.
Loop table/array definition mismatch	A for/loop may define an array or table implicitly though the assigned expressions. If there is a mismatch in shape, indices or index sets which prevents the creation of

	structural arrays on the left hand side of expressions, this error will appear.
Duplicate sets or indices in JSON loop	If a loop contains multiple indices, the indices must be different, i.e. for i in 'parts', j in 'prods')
Loop defined in a wrong section	Loop defined in a wrong section. Loops may only appear in the formulas and preprocessor sections.
Unsupported nesting of loops	Nesting of loops is not allowed or is not allowed in this section.
Incompatible worksheet definitions; check your worksheets:[] and activeSheet: "" properties	The two properties "worksheets" and "activeSheet" are optional properties. However, if one exists, so must the other. The activeSheet: name should be among the names listed in worksheets:[].
RASON can't handle worksheets in names	RASON does not support the use of worksheet names. For example, Con: {formula: "2 * SUM(sheet1!A1:A5)", equal: 2} The use of "sheet1!" is not supported.
RASON can't handle dimensions/cubes	RASON does not support the use of dimensions or cubes.
RASON can't handle Excel TABLE & structured references	RASON does not support the use of Excel tables and Excel structured references. (A combination of table and column names is referred to as a structured reference in Excel, i.e. =SUM(Products[Parts].)
RASON can't handle OFFSET, INDIRECT, VBA functions etc. that required Excel at run-time	RASON does not support the use of the Excel functions OFFSET and INDIRECT or any VBA functions designed by the user.
Invalid parent stage	When the stage-binding value references a non-existing stage. For example, "optStage.number_to_build.finalValue", where "optStage" does not exist.
Circular reference of a parent stage	Similar to circular references with formulas in Excel. If a dependency chain of stages is started, a given stage may not be dependent on a stage that already exists in the chain.
Only one formula def is allowed within an aggregation loop when the loop is part of a sequential code.	If contained within a loop, the operators "sum", "min" and "max" may only contain one formula definition.
More than one formula assigned to an identifier	Only one formula may be attached to a parameter or variable name. For example: A1: {formula: "a formula"} may only be defined once within a RASON model.
Missing constraint/objective type definition	Your RASON optimization model must contain at least one constraint or an objective.
Error in indentParams: [] model definition	Only one formula marked by indentParams:[] is allowed for this type of analysis.
Error in plotParams: [] model definition	A model setting pertaining to sensitivity analysis is not correct, i.e. the property 'sensitivityPoints' should not be < 2 or the property 'sensitivityPoints2' should not be < 0 or = 1.
Invalid Statement Definition	Syntax error in statement definition in { If-then-else, while, Loop, For, sum, min, max }

Statement defined in a wrong setting	Statements may not be defined in all sections, i.e. Loop may not appear in objective/constraints sections.
Error in if-then-else definition	Syntax error in definition of If-then-else statement.
Operation available only with assigned formulas	Operations such as +, - within a formula will execute *only* when the formula is assigned to a cell. For example, if we sum two arrays in a non-array formula, the result depends on the formula cell address. Only applies when a formula is attached to a variable which is an array or contains a reference to another variable or cell.
Missing workflow (DAG) property	A decision flow must be named using the properties flow, flowName, workflow or workflowName. Naming property MUST appear on the first line of the workflow.
Input sources/parameters in models/stages must be unique	Input parameters/data-sources must be unique. This error is returned when two input sources/parameters are given the same name.
Invalid output/result in stage binding	Result entered for input param value or data-source selection does not exist.
Invalid input parameter binding	Input parameter value is set to a mismatched type or dimensions.
Invalid or unsupported model section for the current solving action	Inappropriate section is present in a given model type. For example, the datasets section should not exist in an optimization or simulation model.

The error messages below are specific to RASON Data Mining.

Invalid datasource type. Supported types: csv, xml	Indicates data source is an unsupported type. Currently, RASON Data Mining supports the following file types: "csv", "json", "xml", "excel", "odbc", "access", "msaccess", "mssql", "oracle", "odata".
Invalid estimator/transformer type	Indicates an error in the estimator or transformer type. Currently, RASON Data Mining supports the following estimator/transformer types: "affinityAnalysis", "bigData", "classification", "clustering", "featureSelection", "regression", "textMining", "timeSeries", "transformation".
Invalid algorithm name	The supported names for each algorithm are listed below. "affinityAnalysis"-- "associationRules" "bigData" -- "sampling"or "summarization" "classification" -- "bagging", "boosting", "decisionTree", "nearestNeighbors", "DiscriminantAnalysis",

	"logisticRegression", "naiveBayes", "neuralNetwork", or "randomTrees" "clustering" --"hierarchical" or "kMeans" "featureSelection" -- "linearWrapping", "logisticWrapping", or "univariate" "regression" -- "bagging", "boosting", "decisionTree", "linearRegression", "nearestNeighbors", "neuralNetwork", or "randomTrees" "textMining" -- "latentSemanticAnalysis" or "tfIdf" "timeSeries" -- "addHoltWinters", "autocorrelation", "autocovariance", "difference", "lagAnalysis", "partialAutocorrelation", "arima", "doubleExponential", "exponential", "movingAverage", "mulHoltWinters", or "noTrendHoltWinters" "transformation" -- "binning", "intervalBinning", "CountBinning", "canonicalVariateAnalysis", "categoryReduction", "factorization", "imputation", "oneHotEncoding", "oversamplePartitioning", "partitioning", "principalComponentAnalysis", "rescaling", "sampling", or "stratifiedSampling"
Invalid use of trainData or validData	This error appears when the trainData and validData properties are used incorrectly. Contact Frontline Solvers Support for more information related to your specific model at support@solver.com.
Invalid or missing action property	This error appears when an invalid action property is passed or when an action property is missing. Contact Frontline Solvers Support for more information related to your specific model at support@solver.com.

Invalid enumeration value assigned to parameter or property	If an invalid enumeration value is assigned to a parameter or property, this error will appear.
	The following properties/parameters accept only the values in their sets. If a user enters something different, this error message will appear.
	"aggregationType" has values { "avg", "max", "min", "stddev", "sum" }
	"binningTypeFeatures" and "binningTypeTarget” have values { "equal_count", "equal_interval", "none" }
	"dataForErrorComputation" has values { "only_train", "only_valid", "train_and_valid" }
	"dataFormat" has values { "csv", "parquet" }
	"dissimilarity" has values { "euclidean", "jaccard", "matching" }

	"hiddenLayerActivation" and “outputLayerActivation" have values { "logistic_sigmoid", "softmax", "tanh" }
	"imputationStrategy" has values { "delete_record", "mean", "median", "mode", "value" }
	"inputDataType" has values { "distance_matrix", "raw_data" }
	"learningOrder" has values { "original", "random" }
	"linkage" has values { "centroid", "complete_linkage", "group_average", "mcquitty", "median", "single_linkage", "ward" }
	"matrixMethod" has values { "correlation", "covariance" }
	"metric" has values { "chi2", "cramersv", "fisher", "ftest", "gainratio", "gini", "kendall", "mutualinfo", "pearson", "spearman", "welch" }
	"normType" has values { "l1", "l2" }
	"partitionMethod" has values { "manual", "random", "sequential" }
	"priorProbMethod" has values { "empirical", "manual", "uniform" }
	"prunedTreeType" has values { "full_grown", "best_pruned", "min_error", "manual" }
	"samplingType" has values { "approximate", "exact" }
	"sortOrder" has values { "descending", "ascending" }
	"stratificationMethod" has values { "equal_size", "proportional" }
	"technique" has values { "adjusted_normalization", "normalization", "standardization", "unit_normalization" }
	"weightingScheme" has values { "equal", "inverse_distance" }
	"weightingSchemeDocument" has values { "binary", "entropy", "gf_idf", "inverse", "normal", "prob_idf" }
	"weightingSchemeNormalization" has values { "cosine", "none" }
	"weightingSchemeTerm" has values { "augnorm", "boolean", "logarithmic", "raw_frequency" }

Invalid use of selectedCols or excludedCols	This error appears when the trainData and validData properties are used incorrectly. Contact Frontline Solvers Support for more information related to your specific model.
Invalid evaluation property	An invalid evaluation property has been input for the specified action. Only string values are supported among the set of all data mining evaluation properties.
Decision Table FEEL Errors	
License limit for numbers of rules in decision table has been reached.	
Decision table general error	Internal error – please contact Technical Support.
Duplicate/missing name of a decision table	"Duplicate" indicates that multiple decision tables have identical names. "Missing" indicates that the table name is missing in the upper left hand corner of the decision table.
Unrecognized hit policy	User has entered an unrecognized hit policy for "hitPolicy" property.
Missing or wrong decision table inputs	Decision tables must have at least one input entered as an array [] with either missing type, standard FEEL type or a set of allowed values.
Missing or wrong decision table outputs	Decision tables must have at least one output entered as an array[] with either missing type, standard FEEL type or a set of allowed values.
Num refTypes must = num inputs + num outputs	The number of elements in refTypes must be equal to numInputs + numOutputs.
Unknown ref type. Valid ref types are: Boolean, number, text, date, time and duration.	An unsupported reference type has been passed for retype:[]. Current supported reference types are: Boolean, number, text, date, time and duration.
Decision table column data type mismatch	Indicates there is a value in the input/output column that has a "type" different from the listed "type" specified for the refTypes property, i.e. if an input of type "string" is entered for an input of type ">10".
Inconsistent decision table input values	The array for inputValues: [] must have the same columns as the array for inputs: []
Inconsistent decision table output values	The array for outputValues: [] must have the same columns as the array for inputs: []
Inconsistent decision table output defaults	The array for defaults: [] must have the same columns as the array for outputs:[].
Missing or wrong decision table rules	A decision table must have at least one rule row.
Inconsistent decision table rules dimensions	The array for rules:[] must hae the same number of columns as inputs + outputs.
Input value not covered by the input entries	All values specified for inputValues must be referenced at least once in the rules. For example, if a decision table exists with the property inputValues:['apples', 'pears'] and no input entry mentions "pears", this error will be returned.

Output entry must be a string or number matching the output type	The output entry must have the same type as the corresponding output listed in refTypes.
FEEL string type is required	
FEEL number type is expected	
FEEL any ' – ' in unary test is expected	
FEEL NOT(value) unexpected value	
FEEL expected range operator [..]	
FEEL invalid path '.' operator	
FEEL invalid unary test	
Unary test must be a string expression	Each unary test specified for the "rules" property must be a string expression.
Output entry not covered by output values	All values specified for outputValues must be referenced at least once in the rules. For example, if a decision table exists with the property outputValues:['apples', 'pears'] and no output entry mentions "pears", this error will be returned.
Decision table arguments mismatched	Number of arguments to PsiDecTable() is incorrect.
Decision table input argument is array, must be scalar	An input argument must be a scalar.
No hit found in decision table	No rule evaluated successfully in the decision table.
Multiple hits not allowed with hit policy 'unique'.	A "unique" hit policy must "hit" evaluating to a unique result. If multiple rules are "hit", an error will be returned.
Different hits must have the same values with hit policy 'any'.	With a hit policy of "any", if any rules overlap, but point to the same result, that unique result is returned.
Hit policy with aggregation requires numerical outputs.	If a hit policy with aggregation is used, such as C+, C#, C< or C>, numerical outputs (rather than strings) must be returned in the result.
Output entry not found in the output domain	This error is returned when an output entry is not included in the outputValues domain. For example, if the property outputValues:['apples', 'pears'] exists in a decision table, but a rule output returns 'cherry'.
Composing Data/Time/Duration in decision table result failed	If using data/time/duration strings in an expression, the relevant function must be used, i.e. the expression 'PT2h' + 'PT3h' is invalid. This expression must be rewritten as: duration('PT2h') + duration('PT3h').
Boxed function definition mismatch. Check its syntax. It must be part of a non-array formula.	PsiDecTable() must be entered in a single cell as a non-array formula. To extract an array result in Excel, use the SPILL feature in Microsoft Excel or use Frontline's PsiCalcValue() formula to retrieve the results. For more information on PsiCalcValue see the Analytic Solver Reference Guide.

Unable to locate parent stage data	Pertains to decision flows. This error occurs when a child stage is unable to locate the result of the parent stage.
Unsupported data format	
Unknown failure has occurred while solving RASON model.	An unknown error has occurred. Please contact Frontline Systems Technical Support for help (support@solver.com).
Terminated by user request	The data mining process has been stopped by the user.
Unsupported action	This error occurs when user types action or estimator are unknown for RASON DM.
Unsupported estimator	This error occurs when user types action or estimator are unknown for RASON DM.
Unsupported transformer	This error occurs when user types action or estimator are unknown for RASON DM.
Unsupported forecaster	This error occurs when user types action or estimator are unknown for RASON DM.
Unsupported predictor	This error occurs when user types action or estimator are unknown for RASON DM.
NoDataProvided	No data provided for action.
Unable to retrieve dataset	RASON DM is not able to locate the specified dataset.
Please parse the DAG model first.	????
Stage XX does not exist	The specified stage is missing from the decision flow.
Pipeline for stage X is empty.	RASON DM is unable to locate the contents for stage X.
Failed to locate or load result	RASON DM is unable to locate the result.
Invalid Response Format. STANDALONE cannot be used with workflow models	Response-format=STANDALONE is not supported when solving a decision flow. Use Response-format=WORKFLOW
Please specify non-empty stage name	Each stage must contain a stageName.
Invalid or unknown model type	RASON DM is unable to determine the model type. Use "modelType"="datamining" to specify that the model is a datamining model.
Reusable model X not found	RASON DM is not able to locate the specified reusable model.