Paper SAS4494-2020

The New solveBlackbox Action in SAS® Optimization 8.5

Ed Hughes, Steve Gardner, Josh Griffin, and Oleg Golovidov, SAS Institute Inc.

ABSTRACT

In SAS® Viya® 3.5, SAS® Optimization 8.5 introduces the solveBlackbox action. This action greatly expands both the means of access to this essential optimization technique and the range of its potential applications. Black box optimization uses innovative methods to find solutions to some of the most challenging optimization problems, in which the functions involved might be nonsmooth, discontinuous, and computationally expensive to evaluate. You can call the solveBlackbox action from programs in the CASL, Python, Lua, Java, and R languages. And because this action uses CASL to define objective functions, there is almost no function it cannot work with; any CASL script can become an objective function of the solveBlackbox action. In addition, this action makes extensive use of multilevel parallelism, enabling you to make the most effective use of distributed computational resources. Distributed computational tasks that are called by this action can in turn call other distributed tasks.

In SAS Viya, black box optimization has been available through the OPTMODEL optimization modeling language in SAS Optimization and is the foundation of the Autotune action set, which is used by SAS® Visual Data Mining and Machine Learning to perform automated algorithm selection and hyperparameter tuning for machine learning models. This paper reviews the capabilities of the solveBlackbox action and explores several examples of its use in a variety of settings.

INTRODUCTION

The solveBlackbox action, which was introduced in SAS Optimization 8.5, uses derivative-free heuristic methods to solve nonlinear optimization problems that cannot be solved successfully with traditional optimization approaches. These methods have been available in previous releases as part of the local search optimization solver, which has been renamed the black-box solver in SAS Optimization 8.5.

This paper begins with a brief overview of black-box optimization and its applications. It then describes the features of the solveBlackbox action and discusses how these features distinguish the solveBlackbox action and its practical applications from those of the black-box solver.

WHAT IS BLACK-BOX OPTIMIZATION?

Traditional optimization approaches rely on assumptions about the functions that are used to create constraints and objective functions. For example, linear programming algorithms assume that all functions are linear, and in quadratic programming the objective is quadratic, and all constraints must be linear. Virtually all traditional optimization techniques assume that the functions involved are continuous and continuously differentiable. At a bare minimum, it's assumed that the functions can be expressed as analytic functions of the decision variables.

In contrast, black-box optimization relies on only one assumption—that any function in the optimization model under consideration can be evaluated at any point that is encountered by the algorithm. Thus, black-box optimization treats any function it deals with as a "black box" that produces a value for any encountered point in via a potentially unknown process.

Because black-box optimization does not rely on the existence of first- or second-order derivatives for the functions it deals with, it's also known as "derivative-free optimization."

By discarding the assumptions that support traditional optimization techniques, black-box optimization can address a much broader range of problems than traditional methods, but neither does it benefit from the simplifying consequences of these assumptions. As such, the use of black-box optimization is generally restricted to problems that contain no more than roughly 100 decision variables.

In addition to solving problems with "poorly behaved" functions that violate assumptions of continuity, differentiability, and so on, black-box optimization can solve other challenging classes of problems, including multi-objective optimization and mixed integer nonlinear optimization. Even though black-box optimization does not produce provably optimal solutions, it can find good solutions to extremely difficult problems without requiring users to devise and implement customized solution algorithms.

WHEN SHOULD YOU USE BLACK-BOX OPTIMIZATION?

The chief benefit of black-box optimization is that it can succeed in identifying a good solution to a problem when traditional optimization solvers cannot. The logical conclusion is that if a traditional optimization solver is applicable to your problem, then that is what you should try first. As in all optimization, your first choice among available solvers should be the one that is most highly specialized to handle your problem. Try black-box optimization if there is no traditional solver available, if a traditional solver fails to produce a good solution, or if you lack the time or the resources to build a customized solution algorithm and you need a good solution quickly.

BLACK-BOX OPTIMIZATION WITH SAS

In SAS Optimization, black-box optimization was originally termed "local search optimization (LSO)" because local search and global search algorithms are used simultaneously in this hybrid parallel solution method. The current solver name shifts the focus from the method of solution to the types of problems this solver addresses. This is consistent with the naming of the other SAS Optimization solvers: Linear Programming, Mixed Integer Linear Programming, Network, and so on.

Black-box optimization has been available since SAS Optimization 8.2 as a solver (originally named the "LSO solver") that can be called by the OPTMODEL procedure and, since SAS Optimization 8.3, the runOptmodel action in the Optimization action set. This solver executes multiple instances of global and local search algorithms in parallel. In general, you can regard the black-box solver as specialized type of a genetic algorithm, in which a "population" of solutions evolves over multiple "generations." For details on the black-box solver, see the *SAS® Optimization 8.5 Mathematical Optimization Procedures Guide* (SAS Institute Inc. 2019a).

Black-box optimization is used in SAS® Visual Data Mining and Machine Learning (VDMML) to perform hyperparameter tuning. Every action in the autotune action set relies on black-box optimization to tune hyperparameters—options that govern the training process for predictive models that are produced by machine learning algorithms. Hyperparameter tuning with black-box optimization adjusts these predictive models and as such is a critical element of the machine learning capabilities of VDMML (SAS Institute Inc. 2019c). In this setting, the performance associated with any given set of hyperparameter values is described only by empirical data and there is no expressible objective function, which creates an ideal environment for black-box optimization.

THE SOLVEBLACKBOX ACTION AND THE BLACK-BOX SOLVER

The solveBlackbox action is part of the Optimization action in SAS Optimization. This action enables you to specify optimization problems and solve them with black-box optimization methods (SAS Institute Inc. 2019b). As with the black-box solver, problems can include both linear and nonlinear functions in the constraints and objective function. Decision variables can be real-valued or integer-valued. Single- and multi-objective optimization are both supported.

The solveBlackbox action is not meant to supplant but to complement the black-box solver. Even though both rely on the same solution methods and both can be called from CASL, Python, Lua, Java, or R (the runOptmodel action provides non-CASL access for the black-box solver), there are also important differences between the two.

FUNCTION SPECIFICATION

With the black-box solver you use PROC OPTMODEL syntax to specify the optimization model you are solving. This means you can take full advantage of PROC OPTMODEL's support for index sets that streamline the process of defining optimization model elements and enables you to make model creation data-driven. If you have a lot of experience with PROC OPTMODEL then this is likely to be your preferred modeling approach.

The solveBlackbox action uses CASL syntax to specify an optimization model. You can declare decision variables and specify linear constraints directly with the decVars and linCon parameters, and you specify nonlinear constraints and objective functions indirectly by inputting a block of CASL code via the eval subparameter of the func parameter. The action syntax doesn't explicitly support indexing, but you can create and use indexed data structures in the CASL, R, Python, Lua, and Java languages. If you don't have a lot of PROC OPTMODEL experience but have a good deal of experience in one of these languages, then you will probably find that model creation and model solution with the solveBlackbox action is easier than with the black-box solver.

For example, assume you are programming in Python, and you need to declare the binary decision variables y1 through y8 in building an optimization problem. You could use the decVars parameter of the solveBlackbox action:

```
s.optimization.solveBlackBox(
   decVars = [
      dict(name='y1', lb=0, ub=1, type="I"),
      dict(name='y2', lb=0, ub=1, type="I"),
      dict(name='y3', lb=0, ub=1, type="I"),
      dict(name='y4', lb=0, ub=1, type="I"),
      dict(name='y5', lb=0, ub=1, type="I"),
      dict(name='y6', lb=0, ub=1, type="I"),
      dict(name='y7', lb=0, ub=1, type="I"),
      dict(name='y8', lb=0, ub=1, type="I"),
   ],
   obj = [dict(name='obj', type='min')],
   con = [dict(name='con', ub=1e-3)],
   func = dict(init=caslInit, eval=caslEval),
);
```

If you are an experienced Python programmer, however, you could use a simple loop to accomplish the same declaration:

```
myVars = [];
for i in range(1, 9):
    myVars.append(dict(name='y'+str(i), lb=0, ub=1, type="I"));
```

```
# Invoke the solveBlackbox action
s.optimization.solveBlackBox(
   decVars = myVars,
   obj = [dict(name='obj', type='min')],
   con = [dict(name='con', ub=1e-3)],
   func = dict(init=caslInit, eval=caslEval),
);
```

In this code, you use the decVars parameter simply to reference the code that declares the decision variables (note that range(1,9) halts when i=9 and so only *y1* through *y8* are declared).

The Python code that declares the decision variables is nearly as terse and direct as the corresponding PROC OPTMODEL code:

```
proc optmodel;
   set i=1..8;
   var y{i} binary;
```

The choice between PROC OPTMODEL syntax and Python syntax (or that of another Viya-supported language) thus can be based on programming experience and personal preference and does not inescapably reduce to a choice between terse, data-driven model creation and extensive literal model creation.

MULTILEVEL PARALLELISM

The solveBlackbox action permits you to use multiple levels of parallel execution in defining and solving optimization problems. This enables you to use the solveBlackbox action to create optimization models and solution schemes that aren't possible with the black-box solver and PROC OPTMODEL syntax.

Two parameters of the solveBlackbox action enable you to specify parallel execution of function evaluations on multiple levels. The nParallel parameter specifies how many objective evaluations can be executed in parallel, and nSubsessionWorkers (a subparameter of the func parameter) specifies how many worker nodes to allocate to each subsession that is evaluating an objective function. Together, these parameters enable a scenario where multiple parallel objective function evaluations are themselves executing in parallel (SAS Institute Inc. 2019b).

Recall that you use the eval subparameter of the func parameter to specify objectives and nonlinear constraints for the solveBlackbox action and that you can specify any CASL code block with this subparameter. This includes CASL code that calls other CAS actions, something that cannot be done with PROC OPTMODEL syntax. When you use the eval subparameter along with the nParallel parameter and the nSubsessionWorkers subparameter, you can use the solveBlackbox action to build and solve an optimization problem in which multiple function evaluations execute in parallel and each evaluation executes a CAS action (or several CAS actions) in parallel. This not only dramatically broadens the scope of problems that you can model and solve but also exploits your computational resources more efficiently in doing so.

To illustrate these features, let's consider a case in which an analyst uses Gaussian process regression to fit a set of data. To do this she calls the gpReg action in the Nonparametric Bayes action set (part of SAS Visual Data Mining and Machine Learning). For the purposes of this example, the raw data for the regression consists of 1000 observations from a three-dimensional sine function.

The analyst asks you for help in stress-testing the model before it goes into production use. One approach is to create worst-case performance scenarios for the model by searching for

the maximum error—the difference between the real values and the predicted values. If the computed model error is acceptable, then the model can be deployed with confidence.

The regression model estimates the value of a dependent variable, *y*, as a function of three independent variables *x1*, *x2*, and *x3*. To search for the maximum possible error associated with this model, you can use the solveBlackbox action to solve an unconstrained nonlinear optimization problem that maximizes the absolute value of the error. The following CASL code calls the solveBlackbox action:

```
optimization.solveBlackBox result=r /
         vars = {
            {name='x1',  type='C', lb=0, ub=1},
            {name='x2',  type='C', lb=0, ub=1},
            {name='x3',  type='C', lb=0, ub=1}
          },
         obj = {{name='error', type='MAX'}},
         func = {init=caslInit, eval=caslEval, nSubsessionworkers=2},
         primalOut={name="xsol", replace=true},
         cacheOut={name="xcache", replace=true},
         nParallel = 20,
         popSize=50,
         maxGen=5
        ;
print(r);
run;
```

After declaring the three independent variables, the action call specifies that the objective, error, is to be maximized and designates output CAS tables. The nParallel=20 parameter setting specifies that 20 function evaluations can be done in parallel and the nSubsessionWorkers=2 subparameter setting specifies that 2 worker nodes are to be used for each of these subsessions. Each generation of the genetic algorithm at the heart of the black-box solution method is to contain 50 solutions (popSize=50) and a total of 5 generations are to be created (maxGen=5). The reported value is the maximum error found among all these solutions.

Two CASL code blocks are also specified. The caslInit block is run once for every subsession and loads the appropriate action sets:

```
source caslInit;
      loadactionset 'optimization';
      loadactionset 'aStore';
endsource;
```

The caslEval block specifies how the error objective function is to be evaluated at each point that the solution algorithm encounters:

```
source caslEval;
   /* caslEval is an implicit function that evaluates error(x1,x2,x3)   */
   /* where error(x1,x2,x3) = abs(M(x1,x2,x3) - sin(6*x1+3*x2+1.5*x3))  */
```

```
    /* M(x1,x2,x3) denotes the approx. value returned by the GPreg model */
    /* sin(6*x1+3*x2+1.5*x3) denotes the target function being modeled   */

    /*Create a score table with a single observation*/
    /* Results table with variables x1, x2, x3 */
    columns = {"x1", "x2", "x3"};
    coltypes = {"double", "double", "double"};
    varTable='tempscore';
    sumTab = newtable(varTable, columns, coltypes);
   /* Add a row that will hold values for x1, x2, x3 */
   /* that are generated by solveBlackbox */
    row={x1, x2, x3};
    addrow(sumTab,row);
   /* Save results table to a CAS table */
    saveresult sumTab replace casout=varTable;

   /* Evaluate estimated y value in the regression model */
   /* y_out is a one-observation table with the estimated y value */
    action aStore.score / table={name=varTable}, out={name='y_est'},
    rstore={name='gpregStore'};

   /* Fetch estimated y value from CAS table to a results table */
    table.fetch result=score_table / fetchvars={'P_yMEAN'}
    table={name='y_est'};

    Mx=score_table[1,1].{"P_yMEAN"}[1];
    /* Mx equals M(x), the regression model estimate of y */
    /* Calculate the error */
    f['error'] = abs(Mx - sin(6*x1+3*x2+1.5*x3));
    send_response(f);

   /* Cleanup steps--to keep the next call from failing */
    table.dropTable / table="y_est";

endsource;
```

First, the code stores the values of the independent variables *x1*, *x2*, and *x3* in a single-observation CAS table named varTable. This step uses the `saveresult` action. Next, the `score` action from the aStore action set evaluates the estimated *y* value using the regression model and places it in a CAS table named y_est. The `fetch` action extracts the estimated value of *y* from this CAS table and places it in a results table named score_table

so that it can be used to calculate the model error. Finally, the model error is calculated and is returned. Some final clean-up steps, using the dropTable action, follow.

All four actions that are called here are essential to the objective function evaluation process for this problem. It is the ability of the solveBlackbox action

The key to the flexibility of the objective function evaluation process, which relies on four CAS action calls, is the ability of the solveBlackbox action to use a CASL code block for objective function evaluation. This feature supports a range of problems far beyond what you could accomplish with PROC OPTMODEL syntax.

The full program, including the creation of the source data and the execution of the Gaussian process regression model, is included in the appendix. Recall that what is being estimated here is a sine function, which ranges between -1 and 1. The solveBlackbox action returns a maximum error value of .7532472042. This is cause for great concern. The regression model does not pass its stress test and should be revisited.

QUALITY OF THE IDENTIFIED SOLUTION

Because the solution method that underlies both the solveBlackbox action and the black-box solver is heuristic, it is not guaranteed to identify an optimal solution. Whether you access this solution method via the solver or the action, it will virtually always produce a solution of good quality, but there are cases where there is a distinct difference in solution quality between solver and action.

Consider the following mixed integer nonlinear optimization problem:

$$
\begin{aligned}
&\textit{minimize} && x_1 \cdot x_2 \cdot x_3 \\
&\textit{subject to} && x_1 + (0.1^{y_1}) \cdot (0.2^{y_2}) \cdot (0.15^{y_3}) = 1 \\
& && x_2 + (0.05^{y_4}) \cdot (0.2^{y_5}) \cdot (0.15^{y_6}) = 1 \\
& && x_3 + (0.02^{y_7}) \cdot (0062^{y_8}) = 1 \\
& && y_1 + y_2 + y_3 \geq 1 \\
& && y_4 + y_5 + y_6 \geq 1 \\
& && y_7 + y_8 \geq 13 y_1 + y_2 + 2y_3 + 3y_4 + 2y_5 + y_6 + 3y_7 + 2y_8 \leq 10 \\
& && 0 \leq x_1, x_2, x_3 \leq 1 \\
& && y_1, y_2, \ldots, y_8 \; binary
\end{aligned}
$$

To solve this problem, you could call the runOptmodel action, which submits a block of PROC OPTMODEL code that creates an optimization model and calls the black-box solver:

```
proc cas noqueue;
   loadactionset 'optimization';

   pgm = "
      /* decision variables */
      var x1 >=0  <= 1;
      var x2 >=0  <= 1;
      var x3 >=0  <= 1;
      var y1 binary;
      var y2 binary;
      var y3 binary;
      var y4 binary;
      var y5 binary;
```

```
      var y6 binary;
      var y7 binary;
      var y8 binary;
      /* objective function */
      min f1 =-x1*x2*x3;
      /* constraints */
      con E1: x1+(0.1**y1)*(0.2**y2)*(0.15**y3) = 1;
      con E2: x2+(0.05**y4)*(0.2**y5)*(0.15**y6) =1;
      con E3: x3+(0.02**y7)*(0.06**y8) =1;
      con G4: y1+y2+y3 >= 1;
      con G5: y4+y5+y6 >= 1;
      con G6: y7+y8 >= 1;
      con L7: 3*y1+y2+2*y3+3*y4+2*y5+y6+3*y7+2*y8 <=10;
      /* solve the problem */
      solve with blackbox;";
   runOptmodel / code=pgm;
run; quit;
```

This solution approach finds a solution with an objective function value of -0.668432368.

You could also call the solveBlackbox action and restructure the optimization process. Here, the action selects the values for the binary variables $y_1,\ldots,y_8$ and then uses the casEval code block (calling the NLP solver via the runOptmodel action) to solve for the continuous variables $x_1,\ldots,x_8$:

```
proc cas noqueue;
   source caslInit;
      loadactionset 'optimization';
   endsource;

   source caslEval;
        pgm = "
           /* decision variables */
           var x1 >=0 <= 1;
           var x2 >=0 <= 1;
           var x3 >=0 <= 1;
           /* read in assigned y values as parameters */
           num y1 = " || y1 || ";
           num y2 = " || y2 || ";
           num y3 = " || y3 || ";
           num y4 = " || y4 || ";
           num y5 = " || y5 || ";
```

```
            num y6 = " || y6 || ";
            num y7 = " || y7 || ";
            num y8 = " || y8 || ";
            /* objective function */
            min f1 = -x1*x2*x3;
            /* constraints */
            con E1: x1+(0.1**y1)*(0.2**y2)*(0.15**y3) = 1;
            con E2: x2+(0.05**y4)*(0.2**y5)*(0.15**y6) =1;
            con E3: x3+(0.02**y7)*(0.06**y8) =1;
            con G4: y1+y2+y3 >= 1;
            con G5: y4+y5+y6 >= 1;
            con G6: y7+y8 >= 1;
            con L7: 3*y1+y2+2*y3+3*y4+2*y5+y6+3*y7+2*y8 <=10;
            /* solve the remaining part of the problem */
            solve with NLP;";
   runOptmodel result=runOptmodelResult / code=pgm;

   f['obj'] = runOptmodelResult['objective'];
   f['con'] = runOptmodelResult['infeasibility'];
   send_response(f);
endsource;

/* Use the solveBlackbox action to solve the binary portion      */
/*    and use the caslEval code block to solve the NLP portion   */
/*    for a chosen set of binary decision variable values        */

  optimization.solveBlackbox/
     decVars={
      {name='y1',  lb=0, ub=1, type="I"}
      {name='y2',  lb=0, ub=1, type="I"}
      {name='y3',  lb=0, ub=1, type="I"}
      {name='y4',  lb=0, ub=1, type="I"}
      {name='y5',  lb=0, ub=1, type="I"}
      {name='y6',  lb=0, ub=1, type="I"}
      {name='y7',  lb=0, ub=1, type="I"}
      {name='y8',  lb=0, ub=1, type="I"}
    },
```

```
      obj = {{name='obj', type='min'}},
      con = {{name='con', ub=1e-3}},
      func = {init=caslInit, eval=caslEval},
        popsize = 60,
        maxgen = 20,
        primalout={name="p_out", replace=true},
        cacheout ={name="c_out", replace=true}
      ;
  run;
  quit;


proc print data=sascas1.p_out; run;
```

This modified approach finds a solution with objective function value -0.9434705. This is the true optimal objective function value for this problem. For this problem, segmenting the solution process into black-box and nonlinear phases produces a marked improvement in the quality of the solution that is found (from -0.668432368 to -0.9434705). While it is unlikely that you will be able to predict, for any specific problem, whether this sort of segmented approach will produce a better solution, the solveBlackbox action provides you with the flexibility to pursue it. If you use the black-box solver and you're not satisfied with the solution it produces, you should consider crafting a modified solution approach (as appropriate) with the solveBlackbox action. It might reveal a better solution, or it might confirm a solution you've already found. Either outcome is worthy of the effort involved.

CONCLUSION

The solveBlackbox action is a valuable addition to SAS Optimization. It relies on the same derivative-free heuristic optimization algorithm that supports the black-box solver in SAS Optimization and hyperparameter tuning in SAS Visual Data Mining and Machine Learning. Consequently, the solveBlackbox action can address a broad range of optimization problems that cannot be solved using traditional optimization solution algorithms.

This action adds unique value by expanding access to this solution method beyond the syntax of PROC OPTMODEL. This includes support for native CASL, Python, Lua, Java, and R programming to define optimization problems. The solveBlackbox action also enables you to use other CAS actions to define functions and thus further expands the range of addressable optimization problems. Support for multilevel parallelism can give the solveBlackbox action a computational advantage in solving especially taxing nonlinear optimization problems. Lastly, the solveBlackbox action can in some cases uncover better solutions than those found by other means.

REFERENCES

SAS Institute Inc. (2019a). "*SAS® Optimization 8.5 Mathematical Optimization Procedures.*" Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2019b). "*SAS® Optimization 8.5 Mathematical Optimization Programming Guide.*" Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2019c). "*SAS® Visual Data Mining and Machine Learning 8.5 Programming Guide.*" Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ed Hughes
SAS Institute Inc.
Ed.Hughes@sas.com

APPENDIX: REGRESSION EXAMPLE PROGRAM

```
proc cas;
setsessopt / caslib='CASUSER';
run;
quit;

libname CASUSER cas caslib='CASUSER';

/* create raw data to fit */
data Sinewave;
   keep x1 x2 x3 y;
   call streaminit(123);       /* set random number seed */
   do i = 1 to 1000;
      u = rand("Uniform");
      x1 = u;
      u = rand("Uniform");
      x2 = u;
      u = rand("Uniform");
      x3 = u;
      y = sin(6*x1+3*x2+1.5*x3);
      output;
   end;
   run;

 data casuser.sinewave;
      set Sinewave;
 run;

/* perform Gaussian process regression */
```

```
proc cas;
table.dropTable/ table='gpregStore' quiet=True;
setsessopt / caslib='CASUSER';
loadactionset "nonParametricBayes";
action nonParametricBayes.gpreg result=r ,
   table={name="sinewave"},
   inputs={'x1', 'x2', 'x3'},
   target='y',
   seed=1234,
   nInducingPoints=50, fixInducingPoints=False,
   kernel="RBF",
   partbyfrac = {valid = 0 test=0.2 seed=1235},
   nloOpts={algorithm='SGD',
         optmlOpt={maxIters=320},
         sgdOpt={learningRate=0.01, momentum = 0.9, miniBatchSize=500},
         printOpt={printFreq=10}
        },
   output={casout={name='GpReg_Pred', replace=True}, copyVars="ALL"},
   outInducingPoints={ name="GpReg_inducing", replace=True},
   outVariationalCov={ name="GpReg_S", replace=True},
   savestate={name="gpregStore", replace=True};
print r;
run;
quit;

/* Promote or load store in caslInit */
proc cas;
table.promote / table='gpregStore';
quit;


proc cas noqueue;

   source caslInit;
         loadactionset 'optimization';
         loadactionset 'aStore';
   endsource;
```

```
source caslEval;
      /* caslEval: implicit function that evaluates error(x1,x2,x3) */
      /* error(x1,x2,x3) = abs(M(x1,x2,x3) - sin(6*x1+3*x2+1.5*x3)) */
      /* M(x1,x2,x3) = approx. value returned by the GPreg model */
      /* sin(6*x1+3*x2+1.5*x3) = target function being modeled */

      /*Create a score table with a single observation*/
      /* Results table with variables x1, x2, x3 */
      columns = {"x1", "x2", "x3"};
      coltypes = {"double", "double", "double"};
      varTable='tempscore';
      sumTab = newtable(varTable, columns, coltypes);
     /* Add a row that will hold values for x1, x2, x3 */
     /* that are generated by solveBlackbox */
      row={x1, x2, x3};
      addrow(sumTab,row);
     /* Save results table to a CAS table */
      saveresult sumTab replace casout=varTable;

     /* Evaluate estimated y value in the regression model */
     /* y_out is a one-observation table with the estimated y value */
action aStore.score / table={name=varTable},out={name='y_est'},
rstore={name='gpregStore'};

     /* Fetch estimated y value from CAS table to a results table */
table.fetch result=score_table / fetchvars={'P_yMEAN'}
table={name='y_est'};

      Mx=score_table[1,1].{"P_yMEAN"}[1];
      /* Mx equals M(x), the regression model estimate of y */
      /* Calculate the error */
      f['error'] = abs(Mx - sin(6*x1+3*x2+1.5*x3));
      send_response(f);

     /* Cleanup steps--to keep the next call from failing */
      table.dropTable / table="y_est";

   endsource;
```

```
   /* Programmer is stress-tests the gpreg model */
   /* Runs this optimization to see how poorly it might behave */

   optimization.solveBlackBox result=r /
            vars = {
               {name='x1',  type='C', lb=0, ub=1},
               {name='x2',  type='C', lb=0, ub=1},
               {name='x3',  type='C', lb=0, ub=1}
           },
             obj = {{name='error', type='MAX'}},
             func = {init=caslInit, eval=caslEval, nSubsessionworkers=2},
             primalOut={name="xsol", replace=true},
             cacheOut={name="xcache", replace=true},
             nParallel = 20 /* to speed things up, use more sessions*/,
             popSize=50,
             maxGen=5
            ;
   print(r);
   run;

quit;
```