

Cloud Analytic Services Actions: A Holistic View

Mark Gass, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® Cloud Analytic Services (CAS) provides the analytics that power SAS® Viya®. Just as SAS® provides a wide range of capabilities through its extensive array of procedures, so CAS services are supplied by a wide (and growing) variety of action sets. And just as PROCs each define their own grammar and output tables, so CAS actions define their own programming interface in the form of input parameters and results. But the differences between conventional PROCs and CAS actions are also illuminating. Action execution occurs in parallel on the CAS cluster with reference to a server-side environment (a session, caslibs, tables). And when writing scripts or applications to invoke CAS actions, SAS and other popular industry programming languages are provided an equal footing—with each programming language client customized for its own ecosystem. This paper is a complete introduction to CAS actions and explains how actions execute; introduces the programming concepts common to all actions and to all client languages; and provides a brief tour of the programming interfaces for SAS, Java, Python, R, and REST. A complete understanding of CAS actions will be beneficial to IT as they seek to implement, manage, and operate SAS Viya. And it is essential for SAS Viya programmers, such as data scientists, who want to solve specific problems by invoking CAS analytics directly.

INTRODUCTION

SAS Viya provides an extensive set of analytics tools and applications for the benefit of many types of users, including business users, data scientists, and many types of analytics specialists. The applications include web GUIs for business intelligence, statistics, data mining and machine learning, forecasting, and text analytics. Furthermore, the features provided through SAS Viya GUI applications are equally available through a variety of programming language tools.

CAS is the scalable analytics engine at the heart of SAS Viya. Individual requests to CAS are handled by *actions*. Actions are executable routines that the CAS server makes available to client programs. There are actions for each of the many analytic algorithms, for data management, for administration, and for simple housekeeping.

Since a typical server will host many hundreds of actions, CAS organizes actions into groups – called *action sets*. For example, the tree-based analytics algorithms (decision trees, random forests, and gradient boosting trees) have actions for fitting models, editing models, scoring models, and generating scoring code. All are grouped into the decisionTree action set, which has a total of 12 actions.

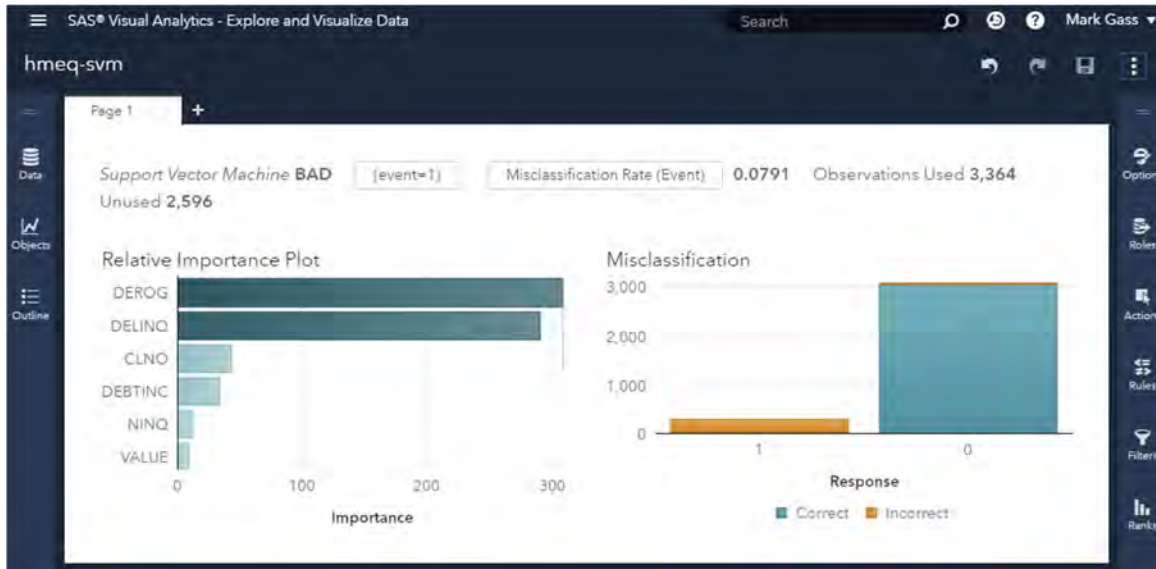
To illustrate the ways that an action can be used, consider building a model for predicting home loan defaults using a Support Vector Machine (SVM). There are many ways to do this in SAS Viya, including the following:

- with the Explore and Visualize Data web GUI from SAS® Visual Analytics
- with the Model Studio web GUI provided by SAS® Visual Data Mining and Machine Learning
- with the SVMACHINE procedure
- with any of the programming language clients

But whichever you choose, you will ultimately run the same action – svmTrain from the svm action set.

The many ways of using SAS Viya are documented in a collection of programming guides and reference manuals on the SAS documentation site “SAS 9.4 and SAS Viya 3.3 Programming Documentation” (SAS Institute Inc. 2017b).

Here is a display illustrating a first attempt to fit an SVM model using SAS Visual Analytics:



Display 1. SVM Model Fit by SAS Visual Analytics – Explore and Visualize Data

This example was fit with the default linear kernel. The high false negative rate of this model might convince you to switch to a cubic kernel. This is an easy change in any of the GUIs or programs. But, rather than exploring SVM hyper-parameter tuning in this paper about CAS actions, we will stick with the default kernel in the examples that follow.

One of the important design principles of SAS Viya is that the analytics are available both in GUI applications and in programming. Furthermore, the programming can take many forms – including both conventional SAS Language and various industry-standard languages.

Thus, SAS Viya makes it convenient for SAS programmers to interact with CAS through the DATA Step and PROCs. Here is the SAS code to build the model using PROC SVMACHINE:

```
data mycas.hmeq;
  set sampsio.hmeq;
run;
proc svmachine data=mycas.hmeq;
  input reason job derog delinq ninq / level=nominal;
  input loan mortdue value yoj clage clno debtinc / level=interval;
  target bad / desc;
run;
```

This code assumes that your SAS session has assigned a libref named “mycas” using the CAS engine. See “Example 18.1: Home Equity Loan Case” (SAS institute Inc. 2017c, p354) if you want the step-by-step details.

HOW ACTIONS EXECUTE

CAS is a server with global state – including data library definitions (caslibs), tables, server options, and libraries of user-defined formats. Global resources are available to all sessions – within the constraints imposed by your site's admins using the CAS authorization system.

To run actions, clients must establish a session using the CAS client programming library for their particular language. (REST is an exception in this regard, as will be described later.) In doing so, the CAS client session creation routine returns a connection to that session, which is then used in future calls.

A session is associated with one user, but a user can have more than one session. Sessions have their own state that is not available to other sessions. This includes session caslibs, session tables (in global or session caslibs), and session options.

Actions have access to the global state and to the state of the session where they are running.

The CAS server isolates sessions in their own OS processes. On an MPP (clustered) CAS server, the session includes an OS process on the controller and each worker. As part of setting up an MPP session, the grid communications subsystem (GC) provides communication among the grid nodes of a session.

At any given time, a server will be running many sessions concurrently. Sessions execute one action at a time. The actions themselves are usually parallelized across their data. Actions know how to find their data blocks across the MPP cluster and how to use multiple cores on each machine in the cluster. Thus, in a cluster with 32 worker nodes and 32 threads per machine, a single analytics action will typically get 1024 threads running in parallel (assuming large enough data to justify that). If a worker fails while an action is executing, the action is restarted with the help of supporting logic from the CAS server.

If an end user or client program wants to run multiple actions in parallel, then multiple sessions are needed. This can occur because the client program drives many sessions – or because an action (like autoTune) itself creates additional sessions.

ACTION INVOCATION SYNTAX

Each action is a self-contained request. After creating a session, client programs make a series of requests to invoke (run) actions on that session. The actions of a particular session execute one at a time.

The grammatical form of an action invocation depends on the client language. For example, an action call is a method call in Java and Python, but in SAS programming it is a statement of the CAS Procedure – which executes the new CAS Language (CASL) scripting language.

The request to execute an action identifies it by name. The action set name is optional – mainly to reduce typing in interactive programming sessions. But an action name might not be unique among all action sets. So, for larger programs, it is best to include both the action set name and the action name to avoid ambiguity.

Here is a simple method invocation in a SAS CASL program.

```
proc cas;
  session casauto;
  action table.caslibinfo / caslib="Public";
run;
```

In SAS programming, "casauto" is the default name for the cas session reference. This "action" statement calls the "caslibinfo" action from the "table" action set and passes the caslib name as a parameter. CAS will respond with the following result:

The SAS System										
Results from table.caslibinfo										
Name	Type	Description	Path	Definition	Subdirs	Local	Active	Personal	Hidden	Transient
Public	PATH	Shared and writeable caslib, accessible to all users.	/opt/sas/viya/config/data/cas/sasmkg/public/		1	0	0	0	0	0

Display 2. Caslibinfo Action Output in SAS

Here is a similar call in Python using the Jupyter notebook. You can see that the action invocation is represented as a method call on an action set object ("table"), which is accessed from the session connection reference ("conn"):

```
In [20]: conn.table.caslibinfo(caslib="HPS")
Out[20]: § CASLibInfo
```

	Name	Type	Description	Path	Definition	Subdirs	Local	Active	Personal	Hidden	Transient
0	HPS	HDFS	HDAT files on /hps	/hps/		1.0	0.0	0.0	0.0	0.0	0.0

```
elapsed 0.00593s - user 0.009s - sys 0.009s - mem 2.28MB
```

Display 3. Caslibinfo Action Invocation and Output in Python

Both of these show the call-by-name approach to identifying parameters. Names used in CAS action calls are not case sensitive in most languages (the Java helper classes for CAS actions are an exception to this rule).

ACTION REFERENCE DOCUMENTATION

Awareness of actions as the building blocks of SAS Viya immediately raises the questions:

- what actions are available?
- what parameters do they accept?
- what information do they return?

The documentation site “SAS 9.4 and SAS Viya 3.3 Programming Documentation” (SAS Institute Inc. 2017b) contains several action set reference guides that answer those questions. This documentation can be accessed in book form (EPUB, and so on), but you will probably prefer the web interface, which lets you focus more quickly on the action at hand and also has quick switching among examples in CASL, Python, Lua, and R.

To find the guides, first navigate to the “SAS Viya Programming” heading and open the sub-heading “CAS Action Programming with CASL, Lua, Python, and R”. Finally, look for the various product-specific sub-headings below that. As of SAS Viya 3.3, these include the following:

- SAS Viya System Programming Guide
- SAS Visual Analytics Programming Guide
- SAS Visual Data Mining and Machine Learning Programming Guide
- SAS Deep Learning Programming Guide
- SAS Visual Text Analytics Programming Guide
- CAS Actions Documented in Other Publications – which links to the following:
 - SAS Optimization: Mathematical Optimization Programming Guide
 - SAS Optimization: Network Optimization Programming Guide
 - SAS Econometrics: Programming Guide
 - SAS Visual Forecasting: Programming Guide
 - SAS Data Quality CAS Action Programming Guide

Basic server interaction is covered in the System Programming Guide. We can look up the “Tables” action set there and find the “caslibInfo” action.

Tables Action Set: Syntax CASL Lua Python R

Provides actions for accessing and managing data

Syntax ▾ Examples ▾ Details

caslibInfo Action
Shows caslib information.

CASL Syntax

```
table.caslibInfo <result=<results> <status=<rc> /
  active=TRUE | FALSE
  caslib="string"
  showHidden=TRUE | FALSE
  srcType="ALL" | "DNFS" | "ESP" | "HDFS" | "LASR" | "PATH" | "S3"
  verbose=TRUE | FALSE
;
```

Parameter Descriptions

active=TRUE | FALSE
when set to True and you do not specify the caslib parameter, information for the active caslib is shown.
Default FALSE

caslib="string"
specifies the name of the caslib to show information for. If not specified, then information for all caslibs is shown.
Alias lib

showHidden=TRUE | FALSE

Display 4. Caslibinfo Action Reference Documentation (CASL Version)

The above documentation shows that the caslibInfo action has several options. One is the name of a specific caslib. It is not required (there is no red asterisk). If omitted, all caslibs would have been listed.

Note the choice in the upper right to display the syntax in CASL, Lua, Python, or R.

Having examined a very simple action to get started, you are now better prepared to look up the reference information about the SVM model-fitting action. Support Vector Machine algorithms are provided by the SAS © Visual Data Mining and Machine Learning product, so that is the documentation set to consult.

Only a few of the most common action sets are loaded into your session when it is created. For others (including svm), your program might need to load the action set. This will be illustrated in the programming examples below.

SAS 9.4 and SAS Viya 3.3 Programming Documentation / SAS Visual Data ... PDF | EPUB

Support Vector Machine Action Set: Syntax CASL Lua Python R

Provides actions for support vector machines

Syntax ▾ Details ▾ Example ▾

Table of Actions

Action Name	Description
svmTrain	Provides actions for support vector machines

Privacy Statement | Terms of Use | Copyright © SAS Institute Inc. All Rights Reserved

Display 5. Locating Support Vector Machine Action Set Reference Documentation

ACTION PARAMETERS

Each action defines a list of (zero or more) parameters. Each parameter has a name and a type. Most client languages encourage a call-by name approach that indicates the name for each parameter along with the value.

Names of action sets, actions, and parameters can sometimes have aliases - but it is best to use the names shown in the documentation.

Parameter lists also have an order – which can sometimes be used to avoid specifying a name – making the call more concise. This is particularly helpful when only the first parameter is passed. For example, in Python, the following uses the more typical named parameter convention:

```
conn.table.droptable(name='CARS')
```

But, in this case, the table name parameter comes first in the list, so the positional parameter approach is possible:

```
conn.table.droptable('CARS')
```

Since most parameters to CAS actions are optional, the by-name approach reduces ambiguity. Names should only be omitted for required parameters that precede all optional parameters. The order of other parameters is not guaranteed over time.

Most importantly, each parameter has an allowed type. This can be a simple scalar value like a floating point number, a string, or an enumeration. The "caslib=" parameter to the caslibinfo action is simply a string.

But quite often, the type is a collection, which can be either of the following:

- an array: a variable-sized list of elements of the same type -or-
- a nested parameter list: which languages might call a “dictionary” or “map”.

And since the elements of these two types of collections might themselves be arrays or parameters lists, the effect is a data structure that can be nested many levels deep.

Client Language	Array Data Type	Array Syntax	Parameter List Data Type	Parameter List Syntax
CASL	Array	['v1', 'v2']	Dictionary	a.b = 'val1'; /*or */ a['c'] = 'val2';
Python	list	['v1', 'v2']	dict	{ 'b': 'val1', 'c': 'val2' }
R	Vector	c('v1', 'v2')	list	list(b='val1', c='val2')
Lua	Table	{ 'v1', 'v2' }	Table	{ 'b'='val1', 'c'= 'val2' }
Java	Array	["v1", "v2"]	Map<String, Object>	actOpts.setB("val1"); actOpts.setC("val2");
REST / JSON	Array	["v1", "v2"]	Object	{ "b": "val1", "c": "val2" }

Table 1. Representation of Complex Types in Various Languages

Finally, common parameter types can be used in different actions. One of the most common examples of that is the "castable" parameter type which you use to identify input tables to many actions.

The various programming guides show common parameter types separately in a section called "Common Parameters". Here is a part of the "castable" documentation:

```
CASL Syntax

* castable={
  caslib="string",
  computedOnDemand=TRUE | FALSE,
  computedVars={{
    format="string",
    formattedLength=integer,
    label="string",
    * name="variable-name",
    nfd=integer,
    nfl=integer
  }}, {...},
  computedVarsProgram="string",
  dataSourceOptions={key-1=any-list-or-data-type-1 <, key-2=any-list-or-data-type-2, ...>},
  groupBy={{
    format="string",
    formattedLength=integer,
    label="string",
    * name="variable-name",
    nfd=integer,
    nfl=integer
  }}, {...},
  groupByMode="NOSORT" | "REDISTRIBUTE",
  importOptions={fileType="AUTO" | "BASESAS" | "CSV" | "DTA" | "ESP" | "EXCEL" | "FMT" | "HDAT" | "JMP" |
    "LASR" | "SPSS" | "XLS", fileType-specific-parameters},
  * name="table-name",
  onDemand=TRUE | FALSE,
  ...
}
```

Display 6. Castable Common Parameter Reference Documentation (CASL Version)

You can see "name=" is a required parameter. On the other hand, "caslib=" is optional because every session has an active caslib, which is chosen if the action invocation does not specify one.

Be aware that most client languages support a concise syntax alternative that sometimes helps with parameter list arguments. If the parameter list has a single required parameter, you can pass that single value instead of using a nested key/value structure. For example, the table parameter specification above is a parameter list supporting quite a few sub-parameters. Here is an example of calling the fetch action – passing input table name and caslib:

```
table.fetch / table={caslib="CASUSER",name="HMEQ"};
```

But if the table is in the session's active caslib, you can allow caslib= to default, so you need only the table name:

```
table.fetch / table="HMEQ";
```

This technique is particularly helpful and widely used when specifying lists of variables to CAS actions.

ACTION RESPONSES

There are essentially two reasons for executing a CAS action. The first is to change the state session or server (loading a table is an example of this). The second is to return information.

Actions return the following information:

- messages
- a final return status
- performance statistics
- a keyed dictionary of result objects

Action Return Status

Several fields are included in the return status for an action, but the most important is the "severity". Severity has values: 0=SUCCESS, 1=WARNING, 2 (or higher) =ERROR. If there is an error, the final error message is also returned in the "status" field.

Output 1 shows a snippet of the SAS Log showing a CASL program executing an errant action and also checking programmatically for its failure. By default, CASL will store the returned status object in the "_status" variable.

```
44     proc cas;
45         session casauto;
46         action table.caslibinfo / caslib="bogus" ;
47         if _status.severity>1 then do;
48             print "action call failed.";
49             print _status;
50         end;
51     run;
NOTE: Active Session now casauto.
ERROR: The caslib 'bogus' does not exist in this session.
ERROR: The action stopped due to errors.
action call failed.
{severity=2,reason=0,status=The caslib 'bogus' does not exist in this
session.,statusCode=2710120}
```

Output 1. Action Return Status in a CASL Program

In this example, the NOTE and ERROR messages were printed by the CASL interpreter, because they were output by the action during its execution. The remaining output comes from the two PRINT statements.

Other languages provide these same four fields as appropriate to their own programming conventions.

Action Results

Actions return a keyed list of results. The means of retrieving them will vary by language.

Each individual result will be one of the following:

- scalar value
- array or parameter list
- a result table

The first two are just as described for parameters – except that rather than being constructed by the client and sent to the server, they are returned from the CAS server to your client program.

The "result table" is actually the most frequently used. For those familiar with SAS programming, it is essentially similar to an ODS output table as illustrated in **Display 2** and Display 3.

It is possible for clients to disconnect from sessions – even while the action is running. Clients normally create a new session when they connect to the server. But it is possible to connect to an existing session. If an action produces results while no client is connected, those results will be queued and returned if a client connects to that session.

DIAGNOSTICS FOR ACTION EXECUTION

A client can list actions previously executed on a session. In SAS, the LISTHISTORY option of the CAS statement accomplishes this.

Output 2 shows the result of listing action history after running PROC SVMACHINE. In the example, the name "casauto" is a reference to the default CAS session:

```
cas casauto listhistory _all_;
<< ... >>
NOTE: 13: action table.tableInfo /
```



```

name='HMEQ', caslib='CASUSERHDFS', quiet=true; /* (SUCCESS) */
NOTE: 14: action table.columnInfo /
table={name='HMEQ', caslib='CASUSERHDFS'},
extended=true, sastypes=false; /* (SUCCESS) */
NOTE: 15: action builtins.loadActionSet / actionSet='svm'; /* (SUCCESS) */
NOTE: 16: action svm.svmTrain /
table={name='HMEQ'},
var={'LOAN', 'MORTDUE', 'VALUE', 'YOJ', 'CLAGE', 'CLNO', 'DEBTINC'},
class={{vars={'REASON', 'JOB', 'DEROG', 'DELINQ', 'NINQ'}}},
emtarget={name='BAD', options={order='FORMATTED', levelType='NOMINAL'}}
; /* (SUCCESS) */

```

Output 2. Using LISTHISTORY to Examine Actions Submitted on a Session

The syntax being printed is that of CASL, which is explained below. (The syntax has been reformatted for better readability.) In this case, it also contains an undocumented "emtarget" option. (SAS PROCs can sometimes generate undocumented options designed specifically for them).

SAS will submit many “bookkeeping” actions as your program interacts with a CAS session. But in this case action call 16 is clearly the one that fits the home loan SVM model.

Output 3 shows a portion of the CAS server, which contains a record of all action calls. There is one line output when the call begins and a later log line for when it completes. The start of action execution is labeled with “++action” followed by the action name. The completion is labeled with “--” and the action name.

Here are the log lines for the request to train an SVM model issued by SAS Visual Analytics. These have been indented and somewhat abbreviated for readability.

```

2017-02-01T17:24:58,351 INFO << ... >> ++ action svm.svmTrain /
table={
  name='HMEQ',
  caslib='Public',
  where=<<no inputs are missing>>,
  computedVars={
    {name='_va_d_BAD_ONES'},
    {name='_EVENT_'},
    {name='_va_FILTER_'}},
  computedVarsProgram='
    '_va_d_BAD_ONES''n=round('BAD''n,1);
    if ((''_va_d_BAD_ONES''n = 1.0))then do;
      '_EVENT_'n= 1.0;
    end;
    else do;
      '_EVENT_'n= 0.0;
    end;
  << ... >>;
',
  computedOnDemand=false, onDemand=false
},
inputs={{name='LOAN'}, {name='MORTDUE'}, {name='YOJ'}, {name='VALUE'},
  {name='CLAGE'}, {name='CLNO'}, {name='DEBTINC'}},
nominals={{name='_EVENT_'}},
target='_EVENT_',
maxiter=25, c=1, noscale=true, tolerance=1E-6,
<< ... >>;
2017-02-01T17:24:58,550 INFO <<...>> -- 'svm.svmTrain' SUCCESSFUL.

```

Output 3. CAS Server Log after SVM Model Fit by SAS Visual Analytics – Explore and Visualize Data

PROGRAMMING LANGUAGE EXAMPLES

With so many client languages to choose from, the paper will not attempt to exhaustively teach the details of programming in each one. So it will not provide step-by-step getting-started information.

Rather it is a survey to help you get the flavor of each language integration – calling out the relative merits and special features of each. So the following sections exhibit a sample of the syntax and link to the detailed documentation.

The References section lists many of the specific documentation resources for programming with CAS actions, but you might find it more convenient to navigate using developer.sas.com (SAS Institute Inc. 2017i), which has sections for each of the CAS client languages.

CASL

You have already seen examples of invoking an action from the SAS language using a PROC SVMACHINE – a PROC specifically developed for SAS Viya. Such PROCs choose the particular actions to run, build the appropriate parameter list, and dissect the action results.

But it is also possible to invoke an action directly from the SAS language using CASL. CASL is a very straightforward scripting language that can be run inside a SAS program by invoking the CAS procedure.

When using CASL, you (as a SAS programmer) have direct control over the CAS actions that are run – making your own choices about parameter values and writing the code to examine each action’s results.

Here is a CASL statement for fitting the home loan SVM model:

```
proc cas;
  action svm.svmTrain result=svmResults /
    table={name='HMEQ', caslib='CASUSER'},
    inputs={'LOAN', 'MORTDUE', 'VALUE', 'YOJ', 'CLAGE', 'CLNO', 'DEBTINC', 'REASON', 'JOB',
'DEROG', 'DELINQ', 'NINQ'},
    nominals={ 'BAD', 'DEROG', 'DELINQ', 'NINQ'},
    target='BAD';
run;
```

The `result=` assigns the complete output of the `svmTrain` action to a variable named “`svmResults`”. Since PROC CAS is an interactive PROC, it is possible to issue a statement to describe the structure of these results:

```
describe svmResults;
run;
```

The Describe statement dumps the full layout of the `svmResults` dictionary into the SAS Log. Output 4 shows how results are a keyed table result objects (highlights added). In this case all the objects are result tables:

```
dictionary ( 6 entries, 6 used);
[ModelInfo] Table ( [8] Rows [4] columns
Column Names:
[1] RowId      [          ] (string)
[2] Descr     [          ] (string)
[3] Value     [          ] (string)
[4] NValue    [          ] (double)
[NObs] Table ( [2] Rows [2] columns
Column Names:
[1] Descr     [          ] (string)
[2] N         [          ] (double)
[TrainingResult] Table ( [10] Rows [3] columns
Column Names:
[1] RowId     [          ] (string)
[2] Descr     [          ] (string)
[3] Value     [          ] (double)
[IterHistory] Table ( [22] Rows [3] columns
Column Names:
[1] Iteration [          ] (double)
[2] Complementarity [          ] (double)
[3] Feasibility [          ] (double)
[Misclassification] Table ( [3] Rows [4] columns
Column Names:
[1] Observed  [          ] (string)
[2] PredEvent [0          ] (double)
[3] PredNonEvent [1          ] (double)
[4] TotalTrain [Total      ] (double)
[FitStatistics] Table ( [4] Rows [2] columns
```

```

Column Names:
[1] Statistic      [          ] (string)
[2] Training       [          ] (double)

```

Output 4. Using CASL DESCRIBE to Dump the Structure of Action Results

Thus, we learn that the results include six different output tables and get lists of each of their columns.

As an extension of the SAS language, CASL is nicely integrated with ODS output. We can ask to print the ones we are interested and obtain ODS output.

```

print svmResults.modelInfo svmResults.misclassification
      svmResults.fitStatistics;
run;

```

The SAS System

modelInfo: Results from svm.svmTrain

Model Information	
Task Type	C_CLAS
Optimization Technique	Interior Point
Scale	YES
Kernel Function	Linear
Penalty Method	C
Penalty Parameter	1
Maximum Iterations	25
Tolerance	1e-06

Misclassification Matrix			
Observed	Training Prediction		
	0	1	Total
0	3055	9	3064
1	257	43	300
Total	3312	52	3364

Fit Statistics	
Statistic	Training
Accuracy	0.9209
Error	0.0791
Sensitivity	0.9971
Specificity	0.1433

Display 7. ODS Output of svmTrain Action Results Printed by PROC CAS PRINT

SAS PROCs are so convenient that you will often prefer to use them if they can handle the problem at hand. But given the large number of actions and extensive choices in their parameters, you might not always find a PROC to make the action calls you want. If not, the option of programming in CASL ensures that all the capabilities of CAS can be conveniently controlled from a SAS program.

Note that as of SAS Viya 3.3, CASL programs can also be submitted to run inside the CAS server itself.

The SAS Viya 3.3 documentation set has both a short getting-started guide (SAS Institute Inc. 2017d) and an extensive reference guide (SAS Institute Inc. 2017e) for CASL.

PYTHON

Python has become very popular among data scientists due partly to the language itself, partly to the accompanying libraries (like Pandas and NumPy), and partly due to related development tools (such as the Jupyter Notebook). The Python interface to SAS Viya is designed for data scientists who have become comfortable and productive in the Python ecosystem. They can add to their toolbox the extensive array of *scalable* analytics offered in SAS Viya.

The Python interface to SAS Viya is supported by a library called SWAT (“Scripting Wrapper for Analytics Transfer”) – an open-source library designed to integrate CAS action programming into Python. Using SWAT, CAS action calls are made via Python methods, and action results are returned as Python dictionaries. Furthermore, the result tables returned by CAS actions and the tables loaded into the CAS server itself can both be accessed as Pandas DataFrames. And Python SWAT is sensitive to the conventions of programming tools like the Jupyter Notebook.

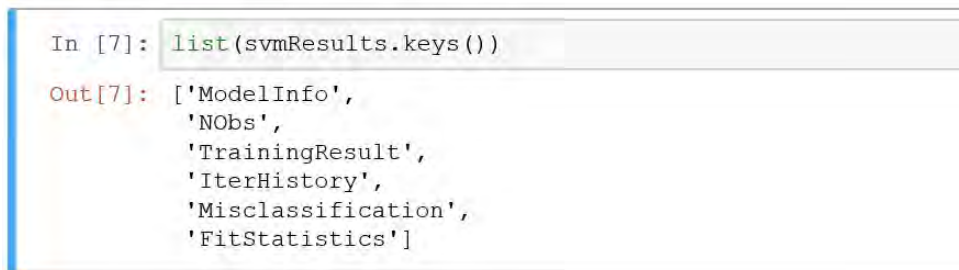
Let’s start with some basic setup to create a session on the server, and then use the connection to that session to load the HMEQ data.

```
import swat
conn = swat.CAS('casa.yourcompany.com', 5570)
conn.table.loadtable(
    caslib='PUBLIC',
    path="hmeq.sashdat",
    casout=dict(caslib='PUBLIC',name="HMEQ"))
```

Once you have a session connection and the home loan data is loaded, you need method calls on the session to load the SVM action set and fit the model.

```
conn.builtins.loadactionset(actionset='svm')
svmResults = conn.svm.svmTrain(
    table=dict(name='HMEQ', caslib='PUBLIC'),
    inputs=['LOAN', 'MORTDUE', 'VALUE', 'YOJ', 'CLAGE', 'CLNO',
            'DEBTINC', 'REASON', 'JOB', 'DEROG', 'DELINQ', 'NINQ'],
    nominals=[ 'BAD', 'DEROG', 'DELINQ', 'NINQ'],
    target='BAD')
```

Now you can dissect the svmResults dictionary. First find out what keys were returned. These correspond to the output tables created by the action.



```
In [7]: list(svmResults.keys())
Out[7]: ['ModelInfo',
         'NObs',
         'TrainingResult',
         'IterHistory',
         'Misclassification',
         'FitStatistics']
```

Display 8. Examining Keys of an SVM Result Dictionary in Jupyter Notebook

Now you might want to print the TrainingResult output table:

```
In [15]: svmResults['TrainingResult']
```

```
Out[15]: Training Results
```

	RowId	Descr	Value
0	WW	Inner Product of Weights	19.800032
1	Beta	Bias	1.537293
2	TotalSlack	Total Slack (Constraint Violations)	532.923480
3	LongVector	Norm of Longest Vector	2.721952
4	nSupport	Number of Support Vectors	3361.000000
5	nSupportInM	Number of Support Vectors on Margin	267.000000
6	MaximumF	Maximum F	2.999994
7	MinimumF	Minimum F	-1.000087
8	nEffects	Number of Effects	12.000000
9	nLevels	Columns in Data Matrix	49.000000

Display 9. An SVM Output Table Displayed by Jupyter Notebook

And, finally, if you wanted to pick out an individual value for later use in your program, you can treat this table as a Pandas DataFrame. The following code applies a filter to pick out a particular row:

```
In [13]: tr =svmResults['TrainingResult']
         tr[tr.RowId=="nEffects"]
```

```
Out[13]: Training Results
```

	RowId	Descr	Value
8	nEffects	Number of Effects	12.0

Display 10. Treating an Output Table as a Pandas DataFrame

The full complement of available programming techniques is too extensive to cover in this paper. If you are a Python programmer, “SAS Viya: The Python Perspective” (Smith and Meng 2017) is a great resource to explain all the features of Python SWAT.

R

R is an open-source statistics programming language with a long legacy in UNIX and PC environments. The SAS Viya CAS client interface to R makes it possible for R users to access the extensive array of grid-scale analytics offered in SAS Viya.

As with Python, there is a SWAT open-source package from SAS that provides the glue.

CAS actions can be called using R functions. When constructing parameter lists, arrays of like values can be passed as R vectors. Nested CAS parameters lists are represented using R's named lists. Also similar to Python, CAS result tables are represented as R data frames.

Assuming that the hmeq data set is loaded as a global table in the CAS server, the following R program fits a home loan default model:

```

library(swat)
conn <- CAS('casa.yourcompany.com', 5570)
loadActionSet(conn,actionSet='svm')

cas.svm.svmTrain(conn,
  table=list(name="hmeq", caslib="public"),
  target="bad",
  inputs=c("loan", "mortdue", "value", "yoj", "clage",
           "clno", "debtinc", "reason", "job", "derog",
           "delinq", "ning"),
  nominals=c("bad", "derog", "delinq", "ning" )
)

```

Some functions that you use are SWAT's own routines provided as programming helpers and others are functions generated by SWAT to wrapper the actions on the server side. SWAT's own routines are documented in the R-SWAT API manual (SAS Institute Inc. 2017). The `loadActionSet` function is a good example of this. It is a key function that does two things:

- calls the CAS builtins.loadActionset action, which loads an optional action set into your session on the server side.
- creates wrapper R functions for every action in the loaded action set.

The `cas.svm.svmTrain` is a good example of a function that SWAT creates to wrapper CAS action execution. This call also demonstrates the use of both CAS array parameters ("inputs" and "nominals" passed as R vectors) and a nested CAS parameter list ("table" passed as an R list with named elements).

Output 5 show some of the output from the above program. This illustrates the printing of output tables, which are R Data Frames.

```

NOTE: SVM training is activated.
$ModelInfo
      RowId      Descr      Value  NValue
1  TaskType      Task Type      C_CLAS    NaN
2  Method Optimization Technique Interior Point    NaN
3  Scale          Scale          YES      NaN
4  Kernel          Kernel Function      Linear    NaN
5  PenaltyMethod   Penalty Method      C        NaN
6  C               Penalty Parameter      1  1.0e+00
7  MaxIter         Maximum Iterations      25  2.5e+01
8  Tolerance       Tolerance          1e-06  1.0e-06

$NObs
      Descr      N
1  Number of Observations Read  5960
2  Number of Observations Used  3364

$TrainingResult
      RowId      Descr      Value
1  WW          Inner Product of Weights  19.800032
2  Beta        Bias          1.537293
3  TotalSlack Total Slack (Constraint Violations)  532.923480
4  LongVector  Norm of Longest Vector  2.721952
5  nSupport    Number of Support Vectors  3361.000000
6  nSupportInM Number of Support Vectors on Margin  267.000000
7  MaximumF    Maximum F          2.999994
8  MinimumF    Minimum F         -1.000087
9  nEffects    Number of Effects  12.000000
10 nLevels     Columns in Data Matrix  49.000000

```

Output 5. Three SVM Output Tables Printed by R-SWAT

LUA

Lua is a simple but flexible language that is designed for embedding in larger programs. Lua succeeds with a minimalist approach to semantics. It relies almost exclusively on a single collection type - the Lua Table – which is an associative array.

In Lua, actions are called as method invocations from the session connection. The method name is a concatenation of action set name, an underscore, and the action name.

CAS action invocations take advantage of the Lua convention of passing a complete Lua table to a method. All parameters are formed into a single (often nested) Lua table. Thus, CAS action calls typically wrap all their arguments in braces – the Lua syntax for passing a single table to a method. Naturally the result of the invocation is also a Lua table.

Here is an example of how the svmTrain action invocation appears in Lua. The final two lines navigate through the results to print the last line of the iteration history table.

```
swat = require 'swat'
conn = swat.CAS('cas1.yourcompany.com', 5570)
conn.table_loadtable{
  caslib='PUBLIC',
  path="hmeq.sashdat",
  casout={caslib='PUBLIC',name="HMEQ"}}
conn.builtins_loadactionset{actionset='svm'}
svmResults = conn.svm_svmTrain{
  table={name='HMEQ', caslib='PUBLIC'},
  inputs={'LOAN', 'MORTDUE', 'VALUE', 'YOJ', 'CLAGE', 'CLNO', 'DEBTINC',
    'REASON', 'JOB', 'DEROG', 'DELINQ', 'NINQ'},
  nominals={ 'BAD', 'DEROG', 'DELINQ', 'NINQ'},
  target='BAD'}
iter = svmResults['IterHistory']
print(iter[#iter])
```

Output 6 show what this Lua program prints – starting with messages emitted by action execution and then concluding with the output from the print() function call:

```
NOTE: Cloud Analytic Services made the file hmeq.sashdat available as table HMEQ in caslib
PUBLIC.
NOTE: Added action set 'svm'.
NOTE: SVM training is activated.
{"Iteration"]=22, ["Complementarity"]=2.3718275E-7, ["Feasibility"]=1.153522E-12}
```

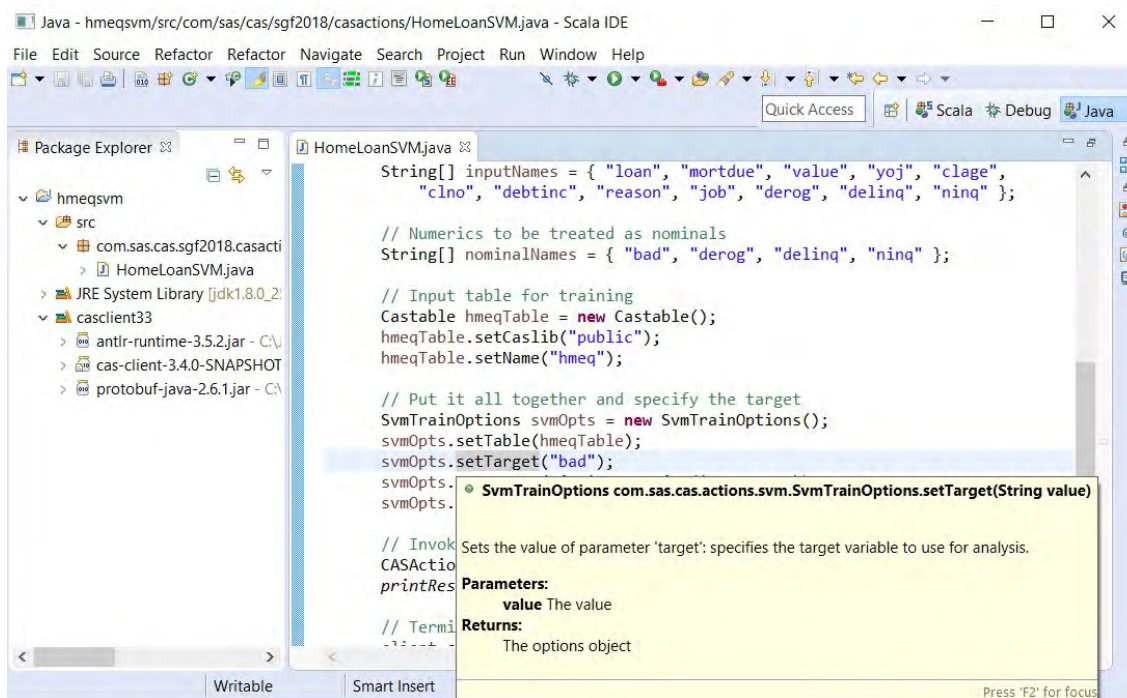
Output 6. Final SVM Iteration Printed by a LUA Script

Given all the other choices available with SAS Viya, Lua is rarely used. But it is required if you are an admin writing a start-up file for the CAS server. And for those who prefer simplicity in their scripting language, Lua is a possible choice.

JAVA (AND SCALA)

By most measures, Java has long been the world's leading programming language. It is particularly popular for mid-tier applications and business applications of all types. The CAS client for Java (cas-client.jar) is a 100% Java implementation, thus fitting smoothly in the Java cross-platform story. The provided Java classes support two styles of interface: strongly typed or named at run time.

The strongly typed approach generates helper classes for all CAS actions and parameter lists. These classes can be particularly helpful if you are programming in an Integrated Development Environment (IDE) like Eclipse. This allows autocompletion and pop-up help as you write your programs.



Display 11. Eclipse IDE with Contextual Pop-ups for Java CASClient

Here is the Java program to fit the home loan default model.

```

package com.sas.cas.sgf2018.casactions;

<<imports omitted>>

public class HomeLoanSVM {

    public static void main(String[] args) {
        String username = args[0];
        String password = args[1];

        System.out.println("CAS SVM Model for Home Loan Default");

        // Start a CAS session.
        CASClientInterface client = new CASClient("cas01", 5570, username,
            password);

        try {
            // Make sure we have the svm action set
            LoadActionSetOptions laOpts = new LoadActionSetOptions()
                .setActionSet("svm");
            client.invoke(laOpts);

            // Fit SVM model for home loan defaults

            // Load the HMEQ table into the session.
            LoadTableOptions ltOpts = new LoadTableOptions();
            ltOpts.setCaslib("Public");
            ltOpts.setCasOut(new Casouttablebasic().setCaslib("Public").setName(
                "HMEQ"));
            ltOpts.setPath("hmeq.sashdat");
            client.invoke(ltOpts);

            // All inputs.
            String[] inputNames = { "loan", "mortdue", "value", "yoj", "clage",
                "clno", "debtinc", "reason", "job", "derog", "delinq", "ninq" };

            // Numerics to be treated as nominals
            String[] nominalNames = { "bad", "derog", "delinq", "ninq" };

```



```

// Input table for training
Castable hmeqTable = new Castable();
hmeqTable.setCaslib("public");
hmeqTable.setName("hmeq");

// Put it all together and specify the target
SvmTrainOptions svmOpts = new SvmTrainOptions();
svmOpts.setTable(hmeqTable);
svmOpts.setTarget("bad");
svmOpts.setInputNames(inputNames);
svmOpts.setNominalNames(nominalNames);

// Invoke the svmTrain
CASActionResults<CASValue> svmResults = client.invoke(svmOpts);
printResults(svmResults);

// Terminate the session and close its socket
client.close(true);
} catch (CASException ce) {
    System.out.println(ce);
} catch (IOException ie) {
    System.out.println(ie);
}
}
return;
}

static CasinvarDesc[] simpleInVarList(String[] names) {
    List<CasinvarDesc> workList = new ArrayList<CasinvarDesc>();
    for (int i = 0; i < names.length; i++) {
        workList.add(new CasinvarDesc().setName(names[i]));
    }
    ;
    return workList.toArray(new CasinvarDesc[0]);
}

static void printResults(CASActionResults<CASValue> results) {
    for (int i = 0; i < results.getResultsCount(); i++) {
        System.out.println(results.getResult(i));
    }
    ;
}
}
}

```

Output 7 shows the beginning from the output (the first result):

RowId	Descr	Value	NValue
TaskType	Task Type	C_CLAS	.
Method	Optimization Technique	Interior Point	.
Scale	Scale	YES	.
Kernel	Kernel Function	Linear	.
PenaltyMethod	Penalty Method	C	.
C	Penalty Parameter	1	1
MaxIter	Maximum Iterations	25	25
Tolerance	Tolerance	1e-06	0.000001
8 rows			

Output 7. Partial SVMTrain Results as Printed from Java

The SAS Viya documentation set includes both a getting-started guide (SAS Institute Inc. 2017g) and reference information in the form of JavaDoc (SAS Institute Inc. 2017h).

Scala has also become popular in machine learning due to its blend of functional and object-oriented programming concepts and also because it runs in the JVM - which is the most common execution


```

        "delinq", "ning"},
    "nominals" : ["bad", "derog", "delinq", "ning"]
}
,

```

CAS responds to this with all the action results information, encoded in JSON. You would then use a JSON parser to pull out the pieces of interest.

Output 9 show the beginning of that output – the first result, which is the FitStatistics output table:

```

{
  "status": 0,
  "log":
    "NOTE: Added action set 'svm'.\nNOTE: SVM training is activated.\n",
  "results": {
    "FitStatistics":
      {
        "_ctb": true,
        "label": "Fit Statistics",
        "title": "Fit Statistics",
        "name": "FitStatistics",
        "schema": [
          { "name": "Statistic",
            "label": "",
            "format": "",
            "type": "string",
            "width": 88,
            "attributes": {}
          },
          { "name": "Training",
            "label": "",
            "format": "",
            "type": "double",
            "width": 8,
            "attributes": {}
          }
        ],
        "attributes":
          {
            "Action": { "type": "string", "value": "svmTrain" },
            "Actionset": { "type": "string", "value": "svm" },
            "CreateTime": { "type": "double",
              "value": 1829399327.84084 },
            "TEMPLATE": { "type": "string",
              "value": "ACAS.SVMACHINE.FitStatistics" }
          }
        },
    "rows": [
      [ "Accuracy", 0.92092746730083 ],
      [ "Error", 0.07907253269916 ],
      [ "Sensitivity", 0.99706266318537 ],
      [ "Specificity", 0.14333333333333 ]
    ]
  },
}
,

```

Output 9. Beginning of REST JSON Response from SVMTrain.

CONCLUSION

From this brief tour, you can see that actions are at the heart of the SAS Viya platform. No matter how you invoke any particular analytic capability, it will be executed by the same action. You have seen how to use the CAS server log and session action history to uncover the actions that SAS Viya applications and PROCs are invoking. And, as you begin to use this diagnostic knowledge to break down particular workloads inside SAS Viya, you can use the action reference documentation to understand the parameters.

Finally, if you want to incorporate the SAS Viya scalable analytics into your own applications or if you want finer control over the steps in your analytics pipeline, then you can choose a programming language and call CAS actions directly.

Given the array of choices, we are often asked which interface or language is best to use. The answer is that it depends on you and your organization. For formal programming projects (ones to be supported by whole teams over a period of time) SAS Viya will work with whatever language your organization has chosen to invest in. For ad hoc work, the answer is to use whatever tools make you personally the most productive – whether that is a GUI, the SAS Language, or one of the other popular programming tools. *The SAS Viya broad array of scalable analytic actions provides the essential value.* The language or user interface you choose is just the gateway to accessing them.

REFERENCES

- SAS Institute Inc. 2017a. SAS Software/python-swat Accessed January 26, 2018. Available: <https://github.com/sassoftware/python-swat>
- SAS Institute Inc. 2017b. *SAS 9.4 and SAS Viya 3.3 Programming Documentation*. Cary, NC: SAS Institute Inc. Available: http://documentation.sas.com/?cdclid=pgmsascdc&cdcVersion=9.4_3.3
- SAS Institute Inc. 2017c. *SAS Visual Data Mining and Machine Learning 8.2: Procedures*. Cary, NC: SAS Institute Inc.
- SAS Institute Inc. 2017d. *Getting Started with CASL Programming 3.3*. Cary, NC: SAS Institute Inc.
- SAS Institute Inc. 2017e. *SAS® Cloud Analytic Services 3.3: CASL Reference*. Cary, NC: SAS Institute Inc.
- SAS Institute Inc. 2017f. “SAS for Developers” Accessed January 26, 2018. Available <https://developer.sas.com>
- SAS Institute Inc. 2017g. *Getting Started with SAS® Viya® for Java*. Cary, NC: SAS Institute Inc.
- SAS Institute Inc. 2017h. “CASClient JavaDoc” Accessed January 26, 2018. Available <https://developer.sas.com/apis/cas/client/java/v3.2.9/apidoc/index.html>
- SAS Institute Inc. 2017i. “SAS For Developers” Accessed January 26, 2018. Available <https://developer.sas.com/>
- SAS Institute Inc. 2017j. *Getting Started with SAS Viya for R*. Cary, NC: SAS Institute Inc.
- SAS Institute Inc. 2017k. SAS Software/R-swat Accessed January 26, 2018. Available <https://github.com/sassoftware/r-swat>
- SAS Institute Inc. 2017l. “SAS Scripting Wrapper for Analytics Transfer API” Accessed January 26, 2018. Available <https://developer.sas.com/apis/swat/r/v1.0.0/R-swat.pdf>
- Smith, K. and Meng, X. 2017. *SAS Viya : The Python Perspective*. SAS Press.

CONTACT INFORMATION

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

Mark Gass
Mark.Gass@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.