Paper SAS3040-2019

What is CASL?

Steven Sober, SAS Institute Inc., Cary, NC

ABSTRACT

This is an introductory paper to help you understand why one would want to adopt the programming language CASL. SAS® Cloud Analytic Services language (CASL) is a language specification used by SAS and other clients to interact with SAS Cloud Analytic Services (CAS). CASL is a statement-based scripting language that supports the entire analytical life cycle (data management, analytics, and scoring), as well as the monitoring of the CAS Server. The strength of this language can be seen in how easy it is to initialize parameters to CAS actions and to manipulate the results of these actions. The results can then be used to prepare the parameters for execution of another action. The goal is to provide an environment that enables a user to string together multiple actions to accomplish a specific result. CASL provides access to popular features in SAS including Base SAS® in-database procedures, DATA step, DS2, FedSQL, formats, and Output Delivery System (ODS). CASL is ideal for splicing together access to the CAS server in between the use of other SAS procedures from a SAS client or from another client, or simply to create your own custom application.

INTRODUCTION

In this paper we will explore how to use the CAS language CASL from a SAS client as well as another client. In addition, we will explore CASL statements that are used to do the following:

1. Load data into CAS
2. Transform the CAS data for analytical processing
3. Model the CAS data
4. Leverage the score code generated by the CASL model

STATEMENT-BASED SCRIPTING LANGUAGE

SYNTAX

CASL is a language specification used by the SAS client and other clients to interact with CAS. CASL is a statement-based language that is case insensitive. CASL is a scripting language that allows us to run actions that create results that SAS and other clients can work with. A strength of CASL is the development of analytic pipelines and running code in CAS with user-defined actions as well as user-defined functions.

Statements can include keywords such as the name of actions and functions, and statements can include expressions. From a SAS client, a single PROC CAS step can contain several CASL programs. From another client, you can interweave non-CASL code, results, and status of the success of the CASL code with non-CASL code.

ACTION SETS

CASL actions are organized with other actions in an action set and usually contain actions that are based on common functionality. In table 1, we see the action sets grouped alphabetically and in table 2, we see the actions sets grouped by products.

Action Sets by Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Action Set	Description
Access Control	Provides actions for managing user access to resources
Aggregate Loss Modeling	Contains actions for modeling aggregate losses with compound distributions
Aggregation	Provides actions for aggregating the values of one or more variables
Analytic Store Scoring	Provides actions for scoring using an analytic store
Association Rule Mining	Provides actions for association rule mining
Audio	Provides actions for processing audio data
Autotune	Provides actions to tune machine learning algorithm hyperparameters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Action Set	Description
Bayesian Network Classifier	Provides actions for performing classification using Bayesian network models
BioMedImage	Provides actions for biomedical image processing
Boolean Rule	Provides actions for Boolean rule extraction and scoring
Builtins	Provides actions that enable server management

Table 1. Actions Sets by Name

Action Sets by Product

SAS Visual Analytics

SAS Visual Statistics

SAS Visual Data Mining and Machine Learning

SAS Visual Text Analytics

SAS Data Quality

SAS Econometrics

SAS Optimization

SAS Visual Forecasting

Table 2. Actions Sets by Product

ACTIONS

CASL actions perform a single task. CAS actions are aggregated with other actions in an action set. You access an action by using the notation `action-set.action`. For example, the table action set contains all of the actions for working with tables, such as adding a table (`table.addTable`), modifying table attributes (`table.attribute`), loading data (`table.loadDataSource`), and dropping a table (`table.dropTable`).

CONNECTING TO CAS

Before we can begin with CASL, we first must understand how to connect to CAS from a client, that is, Python, R, Lua, Java, REST API, and SAS.

SAS CLIENT

From a SAS client we connect to CAS using the following syntax:

```
cas myCAS host="cas.server.com" port=5570;
```

The CAS session name is the 2nd parameter in the above syntax. In our case we are using myCAS, but we can call our CAS session whatever we want. The default value is CASAUTO

OTHER CLIENT

The SAS Scripting Wrapper for Analytics Transfer (SWAT) package allows other clients like Python, R, and Lua to connect to CAS and execute CASL code. The very first step we must take is to download the appropriate SWAT package. Here are the download links for the SWAT packages as well as the syntax to

import the SWAT package and connect to CAS. In all three cases when we connect to CAS we are creating the alias **s** to point to our connection to CAS.

Python SWAT Download

```
In [1]: import swat
In [2]: s = swat.CAS('cas.server.com', 5570, 'username', 'password')
```

R SWAT download

```
library(swat)
s <- CAS("cloud.example.com", 8777, protocol='http')
```

LUA SWAT download

```
> swat = require 'swat'
> s = swat.CAS('cas.server.com', 5570, 'username', 'password')
```

Java - For Java the SWAT package is not used. Instead, access to CAS is by socket protocols and pure Java. For additional information about using Java and REST API to leverage CASL, read the SAS Global Forum 2017 SAS Paper "*I Am Multilingual: A Comparison of the Python, Java, Lua, and REST Interfaces to SAS® Viya®*" by Xiangxiang Meng and Kevin D Smith, SAS Institute Inc.

CASL

Let's take a closer look at a CASL statement. The first example is written from a SAS client. In this example, we are using the action set "table" and its action "loadTable" to load a SAS7BDAT data set called "source_table2.sas7bdat" into the CASLIB "casuser." Caslibs provide a way to organize in-memory tables and an associated data source, that is, Hadoop, Oracle…

SAS CLIENT EXAMPLE

To leverage CASL from a SAS client is very easy. We use the procedure PROC CAS, which ends with a QUIT statement. You can place multiple CASL statements prior to the QUIT statement. We use a RUN statement to execute blocks of CASL code:

```
PROC CAS;
table.loadTable / path="source_table.sas7bdat" casout={caslib="casuser",
                        name="source_table2", replace=true};
RUN;
table.loadTable / path="source_table2.sas7bdat" casout={caslib="casuser",
                        name="source_table2", replace=true};
RUN;
QUIT;
```

PYTHON CLIENT EXAMPLE

Now let's look at the same CASL statement from a Python client. As we can see, we are using the exact same action set, action, and parameters. However, the CASL code is written using the syntax of Python:

```
s.table.loadTable(path="source_table.sas7bdat",
```

```
                  casOut={"caslib":"casuser", "name":"source_table2",
                          "replace":"True"},
                  )
                )
s.table.loadTable(path="source_table2.sas7bdat",
                  casOut={"caslib":"casuser", "name":"source_table2",
                          "replace":"True"},
                  )
                )
```

STATUS

Status provides information about success or failure of the action. To help us understand status, we will use a modeling example using the action `dtreeTrain`, which is part of the `decisionTree` action set. For this example, we will use a SAS client to write our CASL code.

In this example, we are using the `status=rc`; "rc" is the variable containing the severity code. Notice we can use the new variable "rc" in an IF THEN DO block to control if we should continue to execute the code within the IF THEN DO block:

```
proc cas;
    decisionTree.dtreeTrain result=dt status=rc /
      table="iris"
      inputs={"sepallength", "sepalwidth"}
      target="petallength";
run;
   if (0 == rc.severity) then do;
   /* Begin by printing the results. */
      describe dt;
      print dt;
      saveresult dt caslib=casuser casout="irisResults";
   end;
run;
```

The status= parameter returns a severity code. A severity code indicates whether an action succeeded or failed. Table 3 lists the possible values and the meaning of each:

Severity Code	Severity Level Indicated	Description
0	Normal	Indicates that the action completed successfully.
1	Warning	Indicates a minor issue that did not prevent the action from completing successfully. The reason code might provide additional information.

2	Error	Indicates an error that prevented the action from completing successfully. The reason code provides additional information.

Table 3. Severity Code Values and Meanings

A reason code provides general information about the status code. Table 4 lists the possible values and the meaning of each.

Reason Code	Reason Category	Description
0	Success	Indicates that the action completed successfully.
1	Authorization	Indicates a permission problem.
2	Network	Indicates a network problem during connection or a network connection failure.
3	Memory	Indicates an insufficient memory condition.
4	Authentication	Indicates an authentication error.
5	Exception	Indicates an unexpected error condition that prevented the action from completing successfully.
6	Termination	Indicates a severe error condition that causes the action to terminate.

Table 4. Status Reason Code Values and Meanings

DESCRIBE

When results are saved to a variable, the DESCRIBE statement writes a description of the variable to the SAS log. The description shows the expanded arrays and dictionaries to view the depth of the variable.

```
proc cas;
    decisionTree.dtreeTrain result=dt status=rc /
      table="iris"
      inputs={"sepallength", "sepalwidth"}
      target="petallength";
run;
   if (0 == rc.severity) then do;
   /* Describe writes information to the SAS log */
      describe dt;
   /* print utilizes SAS Output Delivery System to publish the results */
      print dt;
      saveresult dt caslib=casuser casout="irisResults";
   end;
run;
```

```
quit;
```

Table 5 shows us the information written to the SAS log when using the DESCRIBE statement. Notice the table name "ModelInfo". The majority of CASL actions return tables, and the DESCRIBE statement is the best way to obtain the table name. The table ModelInfo contains 13 rows and two columns.

```
dictionary ( 1 entries, 1 used);
 [ModelInfo] Table ( [13] Rows [2] columns
Column Names:
 [1] Descr            [               ] (char)
 [2] Value            [               ] (double)

NOTE: The table irisResults has been loaded with 13 observations and 2 variables.
95         quit;

NOTE: PROCEDURE CAS used (Total process time):
      real time           0.37 seconds
      cpu time            0.06 seconds
```

Table 5. Status Reason Code Values and Meanings

RESULTS

As we have seen with the DESCRIBE statement, the results are sent back as an array of dictionaries. A dictionary is associated with a result table to provide additional information about the table. Table 6 contains the dictionary entries.

Property	Description	Syntax
Nrows	This property contains the number of rows in the table.	*result-table*.nrows;
Ncols	This property contains the number of columns in the table.	*result-table*.ncols;
Attrs	This property contains action-specific attributes for the table. This property is not added by all actions.	*result-table*.attrs;
Name	This property contains the name of the table. In most cases, it is the same as the dictionary key that is used to access the table from the results.	*result-table*.name;
Title	This property contains the title of the table.	

Table 6. Result Table Dictionary Entries

When using a SAS client, CASL leverages the SAS Output Delivery System (ODS) to display the results. ODS enables the SAS end users to publish the results in a wide variety of formats. Table 7 displays the results saved to the variable "dt" in the ODS default format:

```
proc cas;
    decisionTree.dtreeTrain result=dt status=rc /
      table="iris"
```

```
      inputs={"sepallength", "sepalwidth"}
      target="petallength";
run;
   if (0 == rc.severity) then do;
   /* Describe writes information to the SAS log */
      describe dt;
   /* print utilizes SAS Output Delivery System to publish the results */
      print dt;
      saveresult dt caslib=casuser casout="irisResults";
   end;
run;
quit;
```

dt: Results from decisionTree.dtreeTrain

Decision Tree for IRIS	
Number of Tree Nodes	33.000000
Max Number of Branches	2.000000
Number of Levels	6.000000
Number of Leaves	17.000000
Number of Bins	20.000000
Minimum Size of Leaves	5.000000
Maximum Size of Leaves	25.000000
Number of Variables	2.000000
Alpha for Cost-Complexity Pruning	0
Number of Observations Used	150.000000
Maximum STD of Leaves	17.373543
Minimum STD of Leaves	1.433318
Mean Squared Error	30.599330

Table 7. Displaying the Results Using SAS Output Delivery System (ODS)

SAVERESULT

Now we will use Python to save our results as well as display the results using pandas. In Python it is very simple to save the results from our decision tree model. We simply assign a new variable. In our case, our new variable is "r"; we can also leverage the status's severity in an IF statement. We do this because we want to print the results and create a pandas DataFrame only if our model ran successfully:

```
r=s.decisionTree.dtreeTrain(
      table="iris",
      inputs={"sepallength", "sepalwidth"},
      target="petallength")


if r.severity == 0:
    print(r)
    df=r['ModelInfo']
```

In table 8 we see the results of the print(r) statement

```
[ModelInfo]

  Decision Tree for IRIS

                                  Descr       Value
0                 Number of Tree Nodes   33.000000
1              Max Number of Branches    2.000000
2                    Number of Levels    6.000000
3                    Number of Leaves   17.000000
4                       Number of Bins   20.000000
5              Minimum Size of Leaves    5.000000
6              Maximum Size of Leaves   25.000000
7                 Number of Variables    2.000000
8   Alpha for Cost-Complexity Pruning    0.000000
9         Number of Observations Used  150.000000
10              Maximum STD of Leaves   17.373543
11              Minimum STD of Leaves    1.433318
12                 Mean Squared Error   30.599330

 + Elapsed: 0.125s, user: 0.0767s, sys: 0.178s, mem: 10mb
```

Table 8. Using Pandas to Display the Results

As we have seen, saving the results in Python is very straight forward. Now let's look at saving the results using a SAS client. In this example, we will save the model results as a CAS table called irisResults:

```
proc cas;
    decisionTree.dtreeTrain result=dt status=rc /
      table="iris"
      inputs={"sepallength", "sepalwidth"}
      target="petallength";
run;
   if (0 == rc.severity) then do;
   /* Describe writes information to the SAS log */
      describe dt;
   /* print utilizes SAS Output Delivery System to publish the results */
      print dt;
      saveresult dt caslib=casuser casout="irisResults";
   end;
run;
quit;
```

Table 9. Using the SAS Client CASL to Save Model Results as the CAS Table irisResults

ANALYTICAL LIFE CYCLE

In this section we explore how to use CASL for analytical data transformations, modeling, and scoring. All CASL examples in this section will be from a Python client. Note: the CASL documentation provides many examples of CASL code using PROC CAS, Python, R, and Lua. In table 9 we see CASL code that has been written using an R client.

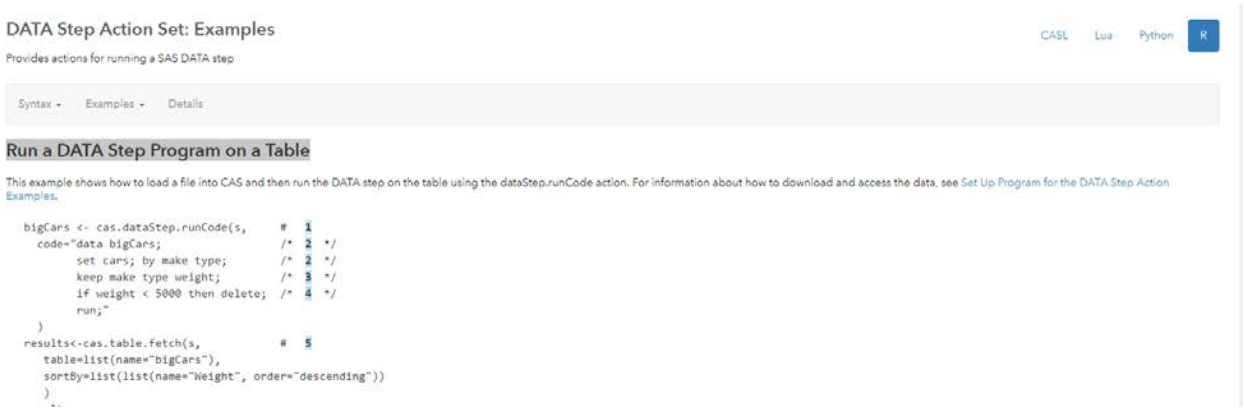

Table 10. CASL Code for Running a DATA Step from an R Client

DATA STEP

One of my favorite things about CASL is that we can leverage DATA step code to transform our data for analytical modeling. The DATA step consists of a group of SAS statements that

begins with a DATA statement and ends with a RUN statement. In its simplest form, the DATA step is a loop with an automatic output and return action.

The DATA Step code in this example is just an example. The code creates new variables and bins the data into categories. What we need to focus on is how to run the DATA step via CASL, not the DATA step code itself. To leverage DATA step we use the "dataStep.runCode" action:

```
# Execute Data Step code
s.dataStep.runCode("""
            data cars_data_step (replace = yes);
                set cars;
                array numvars{2} msrp invoice;
                array sqrtvars{2} sqrt_msrp sqrt_invoice;
                array logvars{2} log_msrp log_invoice;
                array sqrdvars{2} sqrd_msrp sqrd_invoice;
                array invvars{2} inv_msrp inv_invoice;

                do i=1 to dim(numvars);
                sqrtvars{i} = sqrt(numvars{i});
                logvars{i} = log(numvars{i});
                sqrdvars{i} = (numvars{i}**2);
                invvars{i} = (1 / numvars{i});
                end;

                avg_mileage = round(sum(mpg_city, mpg_highway) / 2);

                if 0 <= avg_mileage <= 20 then mileage_cat = 'Poor'; else
                if 21 <= avg_mileage <= 30 then mileage_cat = 'Fair'; else
                mileage_cat = 'Good';

                if 0 <= msrp <= 20000 then class = 'Economy '; else
                if 20001 <= msrp <= 40000 then class = 'Standard'; else
                class = 'Luxury';
            run;
""")
```

DS2

In addition to the DATA step, we can also leverage the SAS language DS2. DS2 is a true object-oriented programing language that can run in CAS as well as in-database, that is,

Hadoop. With DS2 code, we create a THREAD that contains the source data (CARS in this example) and logic to create new variables and bin the data into categories. To leverage the thread to create the output table (CARS_DS2) we will use a DS2 DATA program. In the DS2 DATA program we declare the DS2 THREAD and then execute that THREAD distributed in CAS using a SET statement.

To leverage DS2 in CAS we use the "ds2.runDS2" action:

```
# Run DS2 code via CASL:
s.ds2.runDS2("""
thread mythread / overwrite = yes;
vararray double numvars[2] msrp invoice;
vararray double sqrtvars[2] sqrt_msrp sqrt_invoice;
vararray double logvars[2] log_msrp log_invoice;
vararray double sqrdvars[2] sqrd_msrp sqrd_invoice;
vararray double invvars[2] inv_msrp inv_invoice;
dcl double sqrt_msrp sqrt_invoice log_msrp log_invoice sqrd_msrp sqrd_invoice inv_msrp
inv_invoice avg_mileage;
dcl char(4) mileage_cat;
dcl char(8) class;
method run();
   dcl double i;
   set cars;
   do i=1 to dim(numvars);
   sqrtvars[i] = sqrt(numvars[i]);
   logvars[i] = log(numvars[i]);
   sqrdvars[i] = (numvars[i]**2);
   invvars[i] = (1 / numvars[i]);
   end;

   avg_mileage = round(sum(mpg_city, mpg_highway) / 2);

   if 0 <= avg_mileage <= 20 then mileage_cat = 'Poor'; else
   if 21 <= avg_mileage <= 30 then mileage_cat = 'Fair'; else
   mileage_cat = 'Good';

   if 0 <= msrp <= 20000 then class = 'Economy '; else
   if 20001 <= msrp <= 40000 then class = 'Standard'; else
   class = 'Luxury';
```

```
end;
endthread;

data cars_ds2 / overwrite=yes;
dcl thread mythread t;
method run();
  set from t;
end;
enddata;
""")
```

FEDSQL

In addition to DATA Step and DS2 we can also leverage FedSQL FedSQL is the SAS implementation of the ANSI SQL:1999 core standard. To leverage FedSQL from CASL we use the "fedSql.execDirect" action:

```
# Create a new table using FedSQL
s.fedSql.execDirect(query="""
        create table cars_fedsql{options replace=true} as select
                make, model, mileage_cat, class, avg_mileage, sqrt_msrp, sqrt_invoice,
                log_msrp, log_invoice, sqrd_msrp, sqrd_invoice, inv_msrp, inv_invoice
                from cars_data_step
                where origin = 'USA'
""")

s.fedSql.execDirect(query="""
        create table fact_dim_joined{options replace=true} as select
                a_fact, b_fact, c_fact, a_dim, b_dim, c_dim from fact a join dim b on a.i
= b.i
""")
```

TRANSPOSE

With analytical data preparation we sometimes need to ensure all attributes for a given subject are assigned to that subject. To accomplish this we will use the "transpose.transpose" action:

```
# Transpose data for analytical modeling
```

```
s.transpose.transpose(table={"caslib":"casuser", "name":"cars_fedsql",
"groupBy":[{"name":"mileage_cat"}]},
  transpose={"avg_mileage"},
  id={"model"},
  casOut={"caslib":"casuser", "name":"cars_transposed", "replace":True})
```

MODELING

For our modeling example we will use the logistic regression action. In this example, we are saving the score code generated by the model in "myModel" using the "store=" statement. Note: the store= statement stores the regression models to a blob (binary large object), which we refer to as an "ASTORE", which we can leverage in a 2nd CASL step that is used to score data:

```
m=s.logistic(
      table='source_table',
      classvars='C',
      model={'depvar':'y',
             'effects':{'C', 'x2', 'x8'}},
      store={'name':'myModel', 'replace':'true'},
      output={'casOut':{'name':'out1', 'replace':'true'},
              'pred':'pred', 'resChi':'reschi', 'into':'Predicted Response
Level','copyVars':{'y', 'id'}} )


a=s.fetch(table={'name':'out1','orderby':'id'}, to='5')
print(a)
```

Results from the "print(b)" CASL statement are displayed in table 8.

NOTE: Convergence criterion (GCONV=1E-8) satisfied.
[Fetch]

Selected Rows from Table OUT1

	pred	reschi	Predicted Response Level	y	id
0	0.760728	0.560830	0.0	0.0	1.0
1	0.831197	-2.219025	0.0	1.0	2.0
2	0.903301	-3.056362	0.0	1.0	3.0
3	0.070523	-0.275452	1.0	1.0	4.0
4	0.733170	0.603275	0.0	0.0	5.0

+ Elapsed: 0.0351s, user: 0.0212s, sys: 0.0377s, mem: 7.88mb

Table 11. Results from Modeling the Table source_table

SCORING

Now that we have saved our score code, let's use the "logisticScore" action to score the source table source_table2. Notice we use the "restore=" statement to identify which score (ASTORE) code to use ("myModel"):

```
m=s.logisticScore(
      table='source_table2',
      restore='myModel',
      fitData='false',
      casOut={'name':'out2', 'replace':'true'},
      copyVars={'y', 'id'},
      pred='pred', resChi='reschi', into='Predicted Response Level')
b=s.fetch(table={'name':'out2','orderby':'id'}, to='5')
print(b)
```

In table 12 we can review the results from the "print(b)" statement:

Selected Rows from Table OUT2

	pred	reschi	Predicted Response Level	y	id
0	0.845305	0.427791	0.0	0.0	1.0
1	0.951436	-4.426210	0.0	1.0	2.0
2	0.944953	-4.143216	0.0	1.0	3.0
3	0.279699	-0.623144	1.0	1.0	4.0
4	0.818285	0.471241	0.0	0.0	5.0

+ Elapsed: 0.0347s, user: 0.0167s, sys: 0.0368s, mem: 8.08mb

Table 12. Results from Scoring the Table source_table2

CONCLUSION

CASL is the chameleon of syntax that opens SAS technologies up to the masses. that is, Python, Java, RESTAPI, LUA, R and SAS clients. For over 40 years, SAS has focused on the analytical life cycle. In this paper we explored how CASL enhances this process by providing state-of-the-art technologies for data integration, analytical data transformation, modeling and most importantly the operationalizing of proven models.

ACKNOWLEDGMENTS

I would like to thank Brian Kinnebrew, Bruno Mueller and Jesse Luebbert for their support and input in the creation of this paper.

RECOMMENDED READING

- *SAS Global Forum 2019 Session: 3177 - Got Results? Let CASL Help You Turn them into Superior Reports*

- *SAS Global Forum 2019 Session: 3179 - Can't Find the Right CAS Action? Create Your Own Action Using CASL*
- *SAS® Cloud Analytic Services 3.4: CASL Programmer's Guide*
- *SAS® Cloud Analytic Services 3.4: CASL Reference*
- *SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / Getting Started with CASL*
- *Meng, Xiangxiang, and Kevin D Smith. 2017. "I Am Multilingual: A Comparison of the Python, Java, Lua, and REST Interfaces to SAS® Viya®." Proceedings of the SAS Global Forum 2017 Conference. Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings17/SAS0668-2017.pdf.*

CONTACT INFORMATION

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

Steven Sober
SAS Institute Inc.
919-531-9644
Steven.Sober@sas.com