Paper SESUG 2022-136

Have your cake and eat it too: Automated, sequential SAS batch jobs conditional on programmed log review.

Isaiah Gerber

Abstract: While it is useful to break processing pipelines across multiple programs, at scale it can also make re-running analyses extraordinarily time-consuming and prone to human error. Prior work has tackled this problem by both laying out how to automatically batch numerous SAS programs in sequence (using tools like SAS Enterprise) as well as by developing macros to automate mass log review. This paper explores an approach which allows users to combine these two useful categories of utilities by sequentially batching any number of SAS programs while requiring each program in that sequence to pass a basic log check before batching the next program. The author reviews a simple SAS interface which calls a PowerShell function developed by the author to both 1) automate SAS programs in sequence and to 2) check each resulting log automatically for errors and warnings. This approach allows SAS users to have both the efficiency of automated sequential batching as well as the efficacy of automated log review contained within the same processing pipeline.

INTRODUCTION

While it is best practice to break up complex processing tasks across multiple programs, doing so at scale can also make re-running analyses time-consuming and prone to human error. Traditional batch processing requires a user to batch each individual program in the pipeline, a process that can add significant and unnecessary down time. However, simply automating the entire pipeline runs the risk of data issues being overlooked. This paper elaborates on an approach to this problem that allows interfacing Windows PowerShell with SAS to both automate batch submission while also conditioning each batch run on a log review of the prior program.

QUICK USAGE: SAS_TO_POWERSHELL INTERFACE

```
1
2  /*************************************************************************
3  Program: PowerShell_from_SAS.sas
4
5  Author:  Isaiah Gerber
6
7  Purpose: The program serves as a simple interface to the Start-SAS_Program .ps1
8           program.
9
10 Usage:   To run this program as intended, please update two parameters in the script below:
11            1) Please update the "ps1_dir" to the folder location where you stored the
12               Start-SAS_Program.ps1 file.
13            2) Please update the prg_to_run macro to list the full paths to each program that
14               you wish to batch. Please separate these full paths with either carrage return
15               (seen below) or with whitespace.
16
17 *************************************************************************/
18
19 options XWAIT;
20
21 *Let ps1_dir be the directory in which you stored the Start-SAS_Program.ps1 file;
22 %let ps1_dir=G:\Start-XXX_Program;
23
24 *Let prg_to_run be the list of programs + full filepaths of those programs you wish to batch.
25  Be sure to not include the file extension ("*.sas").;
26 %let prg_to_run=G:\Start-XXX_Program\SAS_testing\1_basic_sas
27                 G:\Start-XXX_Program\SAS_testing\2_basic_sas
28                 G:\Start-XXX_Program\SAS_testing\3_basic_sas;
29
30 *No need to edit this call;
31 x "powershell -file &ps1_dir.\Start-SAS_Program.ps1 &prg_to_run.";
32
```

The SAS_To_Powershell SAS program is designed to provide an effortless way to utilize the Run-SAS_Program.ps1 program without having any knowledge of PowerShell itself. To set up the program, follow these steps:

1. Fill the first macro with the directory where the Start-SAS_Program .ps1 file is stored.
2. Fill the second macro with the list of .sas programs you would like to batch, in the order in which they should be batched.
3. The “x” call to the command line is automatically updated with the macro values you filled in above. There is no need to modify it manually.
4. Run the program! The PowerShell interface will automatically appear once you run the program.

UNDER THE HOOD: THE POWERSHELL SCRIPT

FUNCTIONALITY

The Start-SAS_Program.ps1 file contains two elements: a function definition for a “StartSAS_Program” cmdlet and loop that calls that function on every “argument” specified in the SAS_To_Powershell program. If you call the recall the “prg_to_run” macro that we populated above, then you may have (correctly) guessed that these are the arguments that are being fed through this loop.

The Start-SAS_Program cmdlet can be framed as preforming three distinct operations: running the specified SAS program (the Run-Process cmdlet), reviewing the log for messages (the Select-String cmdlet), and checking to see if the number of messages of a particular type

exceed the user defined threshold (the if/elseif clauses). Each of these is reviewed in turn below:

Run-Process

```
Start-Process SAS -Wait -ArgumentList ("-sysin '" + $Filename + ".sas'
                                       -PRINT '" + $Filename + ".lst'
                                       -LOG   '" + $Filename + ".log'""")
```

The Start-Process cmdlet is a versatile tool within PowerShell which can be used to initiate most programs from the command line. As can be seen from the visual, we feed three arguments into the Start-Process cmdlet:

- The “SAS” argument informs PowerShell that the program we wish to run is a SAS program.
- The “–Wait” option forces PowerShell to not submit any more lines from our PowerShell script until the SAS program is finished batching. This ensures that PowerShell has a populated SAS log to check!
- The “–sysin” option points PowerShell to the specific SAS file to batch.
- The “-PRINT” option specifies the name of the output file from the batch submission.
- The “-LOG” option specifies the name of the log file from the batch submission.

Select-String

```
$note_uninitialized = Select-String -Path ($Filename + ".log") -Pattern 'NOTE: Variable \w* is uninitialized' -Context 1
```

The Select-String cmdlet searches a text document for a string (“-Pattern”) specified by the user. In the case outlined above, we store the results of the Select-String cmdlet in a variable called $note_uninitialized. We feed three arguments into the Select-String cmdlet:

- The “–Path” argument points to the log file that needs to be reviewed. Note that this directory is the same one we specified above for the Start-Process “–LOG” argument.
- The “-Pattern” argument lists the specific string that we wish to pull out of the log file. Note that this argument allows for regular expressions, as can be seen in the example above.
- The “-Context” argument allows us to pull the lines both immediately preceding and following the selected string. A context of 1 pulls the line both immediately preceding and immediately following the selected string, while a context of 5 would pull the 5 previous lines and the 5 following lines.

Conditional statements

```
} elseif ($note_uninitialized.length -gt 0) {
    Write-Host ("The log checker identified $($note_uninitialized.Count) uninitialized variables in " +$Filename + ".log! Processing has stopped. Please review below:")
    $note_uninitialized
    break
```

This series of conditional statements checks to see if the number of notable messages identified in the log check exceeds the user specified tolerance for such messages. This is done as follows:

- We can get at the number of messages identified of a particular type by checking the length of the variable into which we fed the output of the Select-String cmdlet. In the

case above, each row in which an initialized variable was identified corresponds to one observation within the $note_uninitialized variable. We check to see if this number is greater than (“-gt”) 0.

- If the number of identified strings exceeds the specified tolerance, an error message is printed to the PowerShell console (via the Write-Host cmdlet) along with a listing of every string (including context) in which the message was identified. The “break” statement breaks processing out of the function, which stops the processing pipeline.

CURRENT LOG CHECKS

As presented here, the Start-SAS_Program PowerShell program checks for six types of messages within each log file:

1. Errors (0 tolerance)
2. Warnings (1 tolerance)
3. Uninitialized variables (0 tolerance)
4. Character to numeric conversions (0 tolerance)
5. Numeric to character conversions (0 tolerance)
6. Numeric operations performed on invalid data (0 tolerance)

If the number of issues identified exceed their specified tolerance, the program will both stop processing and return the line in the log where the issue was identified.

ADDING LOG CHECKS

While the log checker is currently set up to flag most standard issues, it is also written with customization in mind. If you would like to add an additional check to the log checker, you will need to add two items to the PowerShell code:

1. Add a new “Select-String” cmdlet immediately following the ones already present in the program. This can be written to mimic the ones already programmed, with two exceptions:
    - You will need to change the variable name (the first word, which proceeded by the “$”) to something else, preferably something related to what you are checking for.
    - You will want to change the “-PATTERN” argument to the string you are interested in checking for.
2. Add a new “elseif” bracket block to the program, alongside the other “elseif” blocks. Much as with the “Select-String” cmdlet you added above, this can be written to mimic the other elseif blocks with a few exceptions:
    - Replace the variable name in the condition statement with the name of the variable you specified in the new Select-String cmdlet above
    - Replace the number in the condition statement (the number that follows the “-gt”) with your desired tolerance level. As a reminder, tolerance corresponds to the number of items the message may appear in the log before halts processing.
    - While not strictly necessary for the program to work, it is advised that you change the error message (the text in the “Write-Host” cmdlet) to something that you would find informative for your use case.

LIMITATIONS

Currently, all programs that are processed using this approach can't have any whitespace in their names. For example, while a program named "A_SAS_program.sas" would automate without issue using this approach, a program named "A SAS program.sas" will cause the script to break. This is because PowerShell will treat all characters following the whitespace as options for the command, breaking the script.

CONCLUSION

While it is best practice to split complex processing tasks across multiple programs, doing so at scale can complicate simple processing reruns and unnecessarily bloat the time requirements for task completion. This paper provides one method by which this dilemma may be eased: by automating SAS program batch submission conditional on machine-based log review, it is possible to both maintain the code modularity best practice encourages while also not unnecessarily bogging down the processing timeline.