PharmaSUG 2018 – Paper AD-10

A Method for Independent Program Validation utilising SAS®, R and Python

Aik Hoe Seah, H. Lundbeck A/S

ABSTRACT

The possibilities of comparing results of independent program validation utilising SAS, R and Python were explored. This was achieved utilising the SASPy package (that provides Python APIs to the SAS system), the commonly used data science packages native to Python (NumPY, SciPY, pandas, Matplotlib, statsmodels, etc.), and Rpy2 (that enables users to call R from Python).

INTRODUCTION

In the clinical research industry, program validation is an important quality control (QC) process. For work products that may be less complicated or important, it could be reasonable for a programmer to be the main person programming, while a second programmer will perform the QC. For independent program validations, at least two programmers are assigned to independently develop their programs based on the same set of specifications and the outputs must match to be considered as validated. Although it is alright for the two programmers to use the same programming language while independently developing programs, it is also possible to perform independent coding using different programming languages. The method discussed in this paper will hopefully provide initial ideas in performing independent code validation using different programming environments.

Recent trends in data science has allowed data scientists / analysts to utilize open-source methods to achieve analysis goals. Python, a general use programming language is rising in popularity relative to R and SAS. As statisticians and programmers attempt to decide in the wide variety of choices available for statistical software (SAS / R), Integrated Development Environment (IDEs) and statistical packages, new packages and techniques have emerged from the need to allow communication between each software. This would prove beneficial for our objective to achieve independent program validations in multiple statistical software.

METHODS

To perform the independent program validation using different statistical software, you would require installation of the following listed below:

R-3.x (the latest stable version as of writing is R-3.4.3 for Windows, 6 Dec 2017)

SAS 9.4 or higher (released July 2013) (this is a requirement for SASPy to work)

Anaconda distribution (latest release, as of the publishing of this paper, is version 5.1, 15 Feb 2018) (provides IDEs: Jupyter Notebook, Spyder; and packages: NumPY, SciPY, pandas, Matplotlib)

Install the Python packages statsmodels (this package is typically used to perform statistical analysis native to Python), Rpy2 (to perform statistical analysis in R), SASPy (to perform statistical analysis in SAS). Although detailed documentation and help on how to install the packages can be found on their respective github sites, an example specific to SASPy will be shown in the next section.

As Python is a general use programming language, Python will be used to call R or SAS functions before returning the results to Python as data frames for comparison. It is also possible to try to call R or Python from SAS using IML, or call Python from R using the rPython package. As the scope of this paper only covers calling R or SAS using Python, other possibilities will be explored in the future. Using the Spyder IDE included in Anaconda for Python code development, you can utilise a convenient GUI for developing code. However, if you need to compare results between graphical plots, Spyder's iPython terminal is unable to display the SAS plots generated via SASPy and it is suggested to utilise Jupyter Notebook instead.

INSTALLING AND CONFIGURING SASPY

The easiest way to install SASPy is via pip.[1] On Windows, start up the system shell (cmd.exe), navigate to the location where the Anaconda distribution is installed (using the 'cd' command to navigate folders), and when you have reached the folder where pip.exe exists, key in the following command:

```
pip install saspy
```

Once installed, there are plenty of methods for connecting to SAS, but if you have SAS installed on your local machine, then the easiest would be to use the following configurations to your sascfg.py file:

```
SAS_config_names=['winlocal']
winlocal = {'java'      : 'java',
            'encoding'  : 'windows-1252',
            'classpath' : cpW
            }
```

You may also need to update the paths needed for all the jar files as suggested by the SASPy documentation. For example, the paths for the jar files on my Windows were installed in the following:

```
# Windows client class path
cpW  =  "C:\\Program
Files\\SAS94\\SASDeploymentManager\\9.4\\products\\deploywiz__94360__prt_
_xx__sp0__1\\deploywiz\\sas.svc.connection.jar"
cpW += ";C:\\Program
Files\\SAS94\\SASDeploymentManager\\9.4\\products\\deploywiz  94360  prt
_xx__sp0__1\\deploywiz\\log4j.jar"
cpW += ";C:\\Program
Files\\SAS94\\SASDeploymentManager\\9.4\\products\\deploywiz__94360__prt_
_xx__sp0__1\\deploywiz\\sas.security.sspi.jar"
cpW += ";C:\\Program
Files\\SAS94\\SASDeploymentManager\\9.4\\products\\deploywiz__94360__prt_
_xx__sp0__1\\deploywiz\\sas.core.jar"
cpW += ";
C:\\Users\\xxxx\\AppData\\Local\\Continuum\\Anaconda3\\Lib\\site-
packages\\saspy\\java\\saspyiom.jar"
```

TESTING OUT SASPY ON PYTHON

The next step would be to test it out to see whether Python connects to SAS on your local machine.

```
In [1]: import saspy
sas = saspy.SASsession(cfgname='winlocal')
Out[1]: SAS Connection established. Subprocess id is 7076
```

You have managed to successfully connect to SAS using SASPy and have full functionalities of both SAS and Python.

INSTALLING AND CONFIGURING RPY2

Performing a similar task as SASPy earlier, install and configure Rpy2[2] by launching the system shell (cmd.exe), navigating to pip and entering the following command:

```
pip install rpy2
```

TESTING OUT RPY2 ON PYTHON

If the installation of Rpy2 is successful, you can proceed to test Rpy2. You should have the following results printed (or perhaps any version you have installed).

```
In [2]: import rpy2
print(rpy2.__version__)
Out[2]: 2.8.6
```

RESULTS

Once you have managed to get both SASPy and Rpy2 running, you can start work on this 'multilingual' project. Figure 1 shows the overall view on how results will be compared using the different software.

It is suggested to import datasets created from a single source for all your analyses. Troubleshooting code can be troublesome if the analysis is based on data from different sources and more time would be spent debugging code, if the data wrangling steps are not the same. In this example, Analysis Data Model (ADaM) datasets commonly used in the pharmaceutical industry were used. ADaM is a CDISC compliant data standard used in the clinical research industry for regulatory submissions. A dummy dataset ADVS that contains information about the subjects' Vital Signs was used.

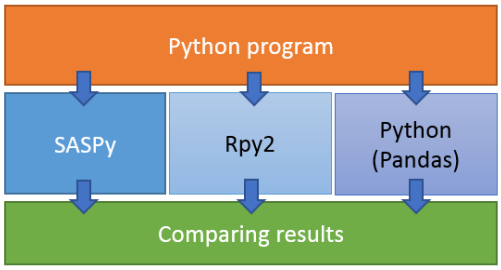

Figure 1. Overall view of independent program validation using Python.

Performing the initial set-ups required for Pandas and importing the ADaM dataset, you can import from a variety of data sources (common file types SAS7BDAT, CSV, TXT are available).

```
In [3]: import pandas as pd
advs_txt = pd.read_table("C://Users//xxxx//studyXYZ//advs.txt")
advs_csv = pd.read_csv("C://Users//xxxx//studyXYZ//advs.csv")
advs_sas = pd.read_sas("C://Users//xxxx//studyXYZ//advs.sas7bdat")
```

You can inspect that the data was read-in correctly as a dataframe.

```
In [4]: advs_sas.head()
Out[4]:
     STUDYID                  USUBJID    BASETYPE      APHASE  NOMWEEK  \
0  b'12345A'  b'12345A-AB1001-S1008'  b'PHASE A'  b'PHASE A'      0.0
1  b'12345A'  b'12345A-AB1001-S1008'  b'PHASE A'  b'PHASE A'     -2.0
2  b'12345A'  b'12345A-AB1001-S1008'  b'PHASE A'  b'PHASE A'      NaN
3  b'12345A'  b'12345A-AB1001-S1008'  b'PHASE A'  b'PHASE A'      NaN
4  b'12345A'  b'12345A-AB1001-S1008'  b'PHASE A'  b'PHASE A'      NaN

      SVDT   ADY  AWTARGET     AVISIT AVISITN FUFL  VSSEQ  AVAL
```

```
0  26986.0   0.0       NaN   b'BASE1'     2.0  NaN   15.0   5.0
1  26973.0 -13.0       NaN  b'SCREEN'     1.0  NaN    9.0   4.0
2  26992.0   6.0       NaN   b'VIS03'     3.0  NaN   21.0   3.0
3  26998.0  12.0       NaN   b'VIS04'     4.0  NaN   27.0   5.0
4  27007.0  21.0       NaN   b'VIS05'     5.0  NaN   33.0   5.0

[5 rows x 38 columns]
```

The data requires some additional steps here; you need to decode some strings with byte encoding to UTF-8.[3]

```
In [5]:
advs_sas['STUDYID'] = advs_sas['STUDYID'].str.decode('utf-8')
advs_sas['USUBJID'] = advs_sas['USUBJID'].str.decode('utf-8')
advs_sas['BASETYPE'] = advs_sas['BASETYPE'].str.decode('utf-8')
advs_sas['PARAMCD'] = advs_sas['PARAMCD'].str.decode('utf-8')
advs_sas['APHASE'] = advs_sas['APHASE'].str.decode('utf-8')
advs_sas['AVISIT'] = advs_sas['AVISIT'].str.decode('utf-8')
advs_sas.head()
Out[5]:
  STUDYID               USUBJID BASETYPE   APHASE  NOMWEEK    NOMWEEKC  \
0  12345A  12345A-AB1001-S1008  PHASE A  PHASE A      0.0   b'Week 0'
1  12345A  12345A-AB1001-S1008  PHASE A  PHASE A     -2.0  b'Week -2'
2  12345A  12345A-AB1001-S1008  PHASE A  PHASE A      NaN         NaN
3  12345A  12345A-AB1001-S1008  PHASE A  PHASE A      NaN         NaN
4  12345A  12345A-AB1001-S1008  PHASE A  PHASE A      NaN         NaN


    ADY  AWTARGET  AVISIT AVISITN FUFL  VSSEQ AVAL
0   0.0       NaN   BASE1     2.0  NaN   15.0  5.0
1 -13.0       NaN  SCREEN     1.0  NaN    9.0  4.0
2   6.0       NaN   VIS03     3.0  NaN   21.0  3.0
3  12.0       NaN   VIS04     4.0  NaN   27.0  5.0
4  21.0       NaN   VIS05     5.0  NaN   33.0  5.0

[5 rows x 38 columns]
```

A few variables in the dataframe has NaN (not a number) values. As these variables were not used for any analysis, they were dropped. You would also need to convert the dataframe to a SAS dataset within SASPy, and to do so via the DF2SD (dataframe to SAS dataset) functionality. Dropping the columns also ensures DF2SD works correctly, as DF2SD is unable to detect the proper format to create the SAS dataset if a variable is completely NaN. You can also print the SAS log to check that the SAS dataset has been created successfully.

```
In [6]:
advs_sas2a = advs_sas.drop(['CRIT1', 'CRIT1FL', 'FUFL'], axis=1)
advs_sas2b = sas.df2sd(advs_sas2a, 'advs', 'WORK')
print(sas.saslog())
```

CREATING SUMMARY TABLES

In this example, the variable CHG can be summarised based on the following conditions: where BASETYPE=”PHASE A” and AVISIT not in (“EARLY1”, “EARLY2”) and using the following keys PARAMCD, AVISITN and AVISIT.

SASPy : SAS code wrapped in Python

```
In [7]:
c1 = sas.submit("""
proc sort data=advs;
    by paramcd avisitn avisit;
run;
proc means data=advs n mean min max;
    where basetype="PHASE A" and avisit not in ("EARLY1", "EARLY2");
    by paramcd avisitn avisit;
    var chg;
    ods output summary=sum1;
run;
""")

print(sas.saslog())
```

There are many procedures that can be used to create summary tables using SAS, one of the most straightforward would be to utilize Proc Means. You can print the SAS log to ensure all the datasets were generated correctly without errors or warnings. The next step would be to convert the SAS dataset 'sum1' generated by the Proc Means back to a Python dataframe. You can do this using SD2DF (SAS dataset to dataframe).

```
In [8]:
advs_sas2out = sas.sd2df('sum1', 'work')
```

Inspecting the new dataframe (shown below), you may notice there is a trailing space found in the PARAMCD and AVISIT variables, which causes issues for comparison later.

```
Out[8]:
  PARAMCD  AVISITN   AVISIT  CHG_N  CHG_Mean  CHG_Min  CHG_Max
0    OPR         1  SCREEN      39  0.205128      -15       14
1    OPR         2   BASE1      40  0.000000        0        0
2    OPR         3   VIS03      40  1.150000      -16       10
3    OPR         4   VIS04      40 -0.225000      -21       11
4    OPR         5   VIS05      35  2.800000       -8       32
```

By removing the trailing space, the dataframe generated via SASPy is complete.

```
In [9]:
advs_sas2out['PARAMCD'] = advs_sas2out['PARAMCD'].str.strip()
advs_sas2out['AVISIT'] = advs_sas2out['AVISIT'].str.strip()
advs_sas2out
Out[9]:
  PARAMCD  AVISITN  AVISIT  CHG_N  CHG_Mean  CHG_Min  CHG_Max
0     OPR        1  SCREEN     39  0.205128      -15       14
1     OPR        2   BASE1     40  0.000000        0        0
2     OPR        3   VIS03     40  1.150000      -16       10
3     OPR        4   VIS04     40 -0.225000      -21       11
4     OPR        5   VIS05     35  2.800000       -8       32
```

Rpy2 : R code wrapped in Python

To run R packages within Python, you can import the following packages with the source data.

```
In [10]:
import rpy2.rinterface as rinterface
import rpy2.robjects as robjects
from rpy2.robjects.packages import importr
rinterface.initr()

base = importr('base')
```

```
sas7bdat = importr('sas7bdat')
robjects.r('advs_r <-
read.sas7bdat("C://Users//xxxx//studyXYZ//advs.sas7bdat")')
```

For the creation of summary tables in R, you utilize the PLYR and DPLYR packages, typically used in R. DPLYR is called after PLYR as some functionality is needed from DPLYR.

```
In [11]:
plyr = importr('plyr')
dplyr = importr('dplyr')
robjects.r('advs_r1 <- select(filter(advs_r, (advs_r$BASETYPE == "PHASE
A") & !(advs_r$AVISIT == "EARLY1" | advs_r$AVISIT == "EARLY2")),
c(1:38))')
robjects.r('advs_r2 <- ddply(advs_r1, c("advs_r1$PARAMCD",
"advs_r1$AVISITN", "advs_r1$AVISIT"), summarise, N = sum(!is.na(CHG)),
mean = mean(CHG, na.rm=TRUE), min = min(CHG, na.rm=TRUE), max = max(CHG,
na.rm=TRUE))')
print(robjects.r('advs_r2'))
```

By inspecting the R dataframe created, you can see that the column labels would need to be updated.

```
Out[11]:
  advs_r1$PARAMCD  advs_r1$AVISITN advs_r1$AVISIT   N      mean   min
max
1             OPR              1.0         SCREEN  39  0.205128 -15.0
14.0
2             OPR              2.0          BASE1  40  0.000000   0.0
0.0
3             OPR              3.0          VIS03  40  1.150000 -16.0
10.0
4             OPR              4.0          VIS04  40 -0.225000 -21.0
11.0
5             OPR              5.0          VIS05  35  2.800000  -8.0
32.0
```

You can convert the R dataframe back to a Python dataframe using the Pandas2RI functionality. Furthermore, you would need to reset the index of the dataframe after updating the column labels. The index needs to be reset as indices of dataframes in R start from 1, while indices of Python dataframes start from 0.

```
In [12]:
from rpy2.robjects import r, pandas2ri
pandas2ri.activate()

r.data('advs_r2')
r['advs_r2'].head()
advs_rout = r['advs_r2']
advs_rout1 = advs_rout
advs_rout1.columns = ['PARAMCD', 'AVISITN', 'AVISIT', 'CHG_N',
'CHG_Mean', 'CHG_Min', 'CHG_Max']
advs_rout1 = advs_rout.reset_index()
advs_rout2 = advs_rout1.drop(['index'], axis=1)
advs_rout2
```

You can check that the ADVS_ROUT2 dataframe is as intended, through visual inspection.

```
Out[12]:
  PARAMCD  AVISITN  AVISIT  CHG_N  CHG_Mean  CHG_Min  CHG_Max
0     OPR      1.0  SCREEN     39  0.205128    -15.0     14.0
1     OPR      2.0   BASE1     40  0.000000      0.0      0.0
2     OPR      3.0   VIS03     40  1.150000    -16.0     10.0
3     OPR      4.0   VIS04     40 -0.225000    -21.0     11.0
4     OPR      5.0   VIS05     35  2.800000     -8.0     32.0
```

Summary tables using Pandas (Python)

To create summary tables via Python, you utilize the Pandas package. Using the ADVS_SAS dataframe imported earlier, ADVS_PY5 dataframe was generated using the GROUPBY and AGG functions.

```
In [13]:
advs_py = advs_sas
advs_py.head()

advs_py1 = advs_py[advs_py['BASETYPE'] =="PHASE A"]
advs_py2 = advs_py1[advs_py1['AVISIT'] !="EARLY1"]
advs_py3 = advs_py2[advs_py2['AVISIT'] !="EARLY2"]

advs_py4 = advs_py3.groupby(["PARAMCD", "AVISITN", "AVISIT"]).agg({"CHG":
['count', 'mean', 'min', 'max']})
advs_py5 = advs_py4
```

Checking the ADVS_PY5 dataframe, you can see that the column labels are not the same as the ADVS_SAS2OUT and ADVS_ROUT2 dataframes.

```
In [14]: advs_py5
Out[14]:
                        CHG
                      count      mean   min   max
PARAMCD AVISITN AVISIT
OPR     1.0     SCREEN    39  0.205128 -15.0  14.0
        2.0     BASE1     40  0.000000   0.0   0.0
        3.0     VIS03     40  1.150000 -16.0  10.0
        4.0     VIS04     40 -0.225000 -21.0  11.0
        5.0     VIS05     35  2.800000  -8.0  32.0
```

Hence, you need to update the column labels.

```
In [15]:
advs_py5.columns = ['CHG_N', 'CHG_Mean', 'CHG_Min', 'CHG_Max']
advs_py5
Out[15]:
                       CHG_N  CHG_Mean  CHG_Min  CHG_Max
PARAMCD AVISITN AVISIT
OPR     1.0     SCREEN     39  0.205128    -15.0     14.0
        2.0     BASE1      40  0.000000      0.0      0.0
        3.0     VIS03      40  1.150000    -16.0     10.0
        4.0     VIS04      40 -0.225000    -21.0     11.0
        5.0     VIS05      35  2.800000     -8.0     32.0
```

Inspecting the shape of the dataframe, you can also see that 5x4 is a result of the GROUPBY and AGG functions applied earlier, and you would like to have 5x7 instead, to match the ADVS_SAS2OUT and ADVS_ROUT2 dataframes.

```
In [16]: advs_py5.shape
Out[16]: (5, 4)

In [17]: advs_sas2out.shape
Out[17]: (5, 7)

In [18]: advs_rout2.shape
Out[18]: (5, 7)
```

You perform the same function to reset the index of the dataframe. Subsequently, you can check that the shape is now at 5x7.

```
In [19]: advs_py6 = advs_py5.reset_index()
advs_py6.shape
Out[19]: (5, 7)

In [20]: advs_py6
Out[20]:
  PARAMCD  AVISITN  AVISIT  CHG_N  CHG_Mean  CHG_Min  CHG_Max
0     OPR      1.0  SCREEN     39  0.205128    -15.0     14.0
1     OPR      2.0   BASE1     40  0.000000      0.0      0.0
2     OPR      3.0   VIS03     40  1.150000    -16.0     10.0
3     OPR      4.0   VIS04     40 -0.225000    -21.0     11.0
4     OPR      5.0   VIS05     35  2.800000     -8.0     32.0
```

COMPARING TABLE OUTPUT RESULTS

If there are only a few outputs to compare results, it could be more efficient to do a manual check. However, it is suggested to perform comparisons programmatically, because it reduces the possibilities of human error while checking.[4] This allows checking to be done across outputs with a lot of values. If you have a lot of summary tables to compare (perhaps in the hundreds or even thousands), it would not be feasible to attempt manual checks as you would need either more resources or time.

Experienced SAS programmers can opt to convert all the dataframes to SAS datasets, and perform their comparison using Proc Compare. If you would like to compare the dataframes using Python, there are a few pointers to take note. There could be minor differences with regards to floating point values when summarized via different environments (R, SAS, Python). While checking for differences, the values could be the same visually, but programmatically, the Booleans performing the comparison could possibly give a True (that there are differences) result instead of being False (no differences).

```
In [21]: (advs_py6 != advs_rout2).any(1)
Out[21]:
0    False
1    False
2    False
3    False
4    False
dtype: bool
```

Using the simplistic approach above, you can compare the dataframe generated via Pandas vs Rpy2, and that there were no differences for all records.

```
In [22]: (advs_py6 !=
advs_sas2out).any(1)
Out[22]:
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

```
In [23]: (advs_rout2 !=
advs_sas2out).any(1)
Out[23]:
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

Comparing the dataframe generated via Pandas vs SASPy however, there were differences detected for one record. And this also happens when you compare the dataframe generated via Rpy2 vs SASPy.

```
In [24]:
ne_stacked = (advs_py6 != advs_sas2out).stack()
changed = ne_stacked[ne_stacked]
changed.index.names = ['id', 'col']
difference_locations = np.where(advs_py6 != advs_sas2out)
changed_from = advs_py6.values[difference_locations]
changed_to = advs_sas2out.values[difference_locations]
```

```
diff02 = pd.DataFrame({'from': changed_from, 'to': changed_to},
index=changed.index)
diff02

Out[24]:
                  from        to
id col
0  CHG_Mean  0.205128  0.205128
```

Further investigation of the differences between the 2 dataframes was not helpful. Visually, the 2 dataframes looked like they matched exactly.

```
In [25]:
advs_py6['CHG_Mean']._data
Out[25]:
SingleBlockManager
Items: RangeIndex(start=0, stop=5, step=1)
FloatBlock: 5 dtype: float64

In [26]:
advs_sas2out['CHG_Mean']._data
Out[26]:
SingleBlockManager
Items: RangeIndex(start=0, stop=5, step=1)
FloatBlock: 5 dtype: float64
```

Checking the floating types did not reveal much information.

As CHG_Mean is the column that keeps tripping up the comparisons, you would need to find out why this is happening and to what degree the differences could be. For this record, the value started to be different at the 14th decimal.

```
In [27]:(advs_py6[0:1] != advs_sas3out[0:1]).any(1)
Out[27]:
2    True
dtype: bool

In [28]:(advs_py6["CHG_Mean"][0:1] !=
advs_sas3out["CHG_Mean"][0:1]).any()
Out[28]: True

In [29]:y=12
(advs_py6["CHG_Mean"][0:1].round(y) !=
advs_sas3out["CHG_Mean"][0:1].round(y)).any()
Out[29]: False

In [30]:y=13
(advs_py6["CHG_Mean"][0:1].round(y) !=
advs_sas3out["CHG_Mean"][0:1].round(y)).any()
Out[30]: False

In [31]:y=14
(advs_py6["CHG_Mean"][0:1].round(y) !=
advs_sas3out["CHG_Mean"][0:1].round(y)).any()
Out[31]: True
```

Incorporating the ability to test if two data frames matched to a certain decimal place, slight modifications were made to the code, and introduced as a function definition. Running the following function in the next page, you can compare two data frames, while handling the possibility that some values could be different at certain decimal places, due to the inherent nature of floating values.

```
In [32]: def compareDF(df1, df2, y):
    ne_stk = (df1.round(y) != df2.round(y)).stack()
    changed = ne_stk[ne_stk]
    if changed.shape==(0,):
        print ("no differences found")
    else:
        changed.index.names = ['variable', 'value']
        diff_locate = np.where(df1.round(y) != df2.round(y))
        changed_from = df1.values[diff_locate]
        changed_to = df2.values[diff_locate]
        diff = pd.DataFrame({'from': changed_from, 'to': changed_to},
index=changed.index)
        return diff

In [33]: compareDF(advs_py6, advs_sas3out, 12)
no differences found

In [34]: compareDF(advs_py6, advs_sas3out, 13)
no differences found

In [35]: compareDF(advs_py6, advs_sas3out, 14)
Out[35]:
                      from        to
variable value
0        CHG_Mean  0.205128  0.205128
```

COMPARING GRAPHICAL PLOT RESULTS

It may not be a trivial task to compare results from graphical plots, especially if there were overlapping data points and lines. The method suggested for comparing plots would be to output the final values used for plotting, into a data set. And then you can perform the same function defined above for comparison between two data frames. If you want to do an eyeball check on the graphs itself, you can create plots using the Matplotlib, Bokeh or Seaborn libraries on Python. If you want to create SAS plots, the Spyder IDE would not suffice and you would have to utilize Jupyter Notebook instead.

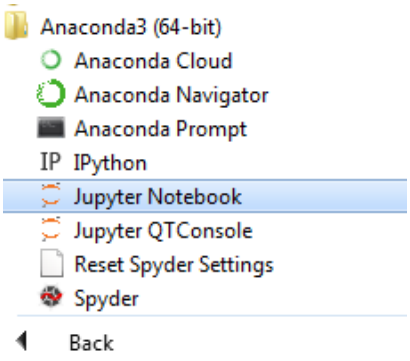

Figure 2. Location of Jupyter Notebook

Jupyter Notebook should have been installed together with Spyder while installing the Anaconda distribution. It can be found by navigating from the Start Menu as shown in Figure 2. In the following example plots generated from the following packages SAS: Proc Sgplot, Python: Seaborn, R: lattice will be compared.

You would need to perform all the steps in loading the data needed into Jupyter, just as what had been done using Spyder earlier. Using Proc Sgplot, you can wrap your SAS code as a Python object, and output the result using the command that prints the LST into HTML.

```
In [1]: import pandas as pd
import numpy as np
from IPython.display import HTML
import saspy
sas = saspy.SASsession(cfgname='winlocal')
advs_sas = pd.read_sas("C://Users//xxxx//studyXYZ//advs.sas7bdat")
advs_sas['STUDYID'] = advs_sas['STUDYID'].str.decode('utf-8')
advs_sas['USUBJID'] = advs_sas['USUBJID'].str.decode('utf-8')
advs_sas['BASETYPE'] = advs_sas['BASETYPE'].str.decode('utf-8')
```

```
advs_sas['PARAMCD'] = advs_sas['PARAMCD'].str.decode('utf-8')
advs_sas['APHASE'] = advs_sas['APHASE'].str.decode('utf-8')
advs_sas['AVISIT'] = advs_sas['AVISIT'].str.decode('utf-8')
advs_sas2a = advs_sas.drop(['CRIT1', 'CRIT1FL', 'FUFL'], axis=1)
advs_sas2b = sas.df2sd(advs_sas2a, 'advs', 'WORK')

In [2]: c = sas.submit("""
proc sgplot data=work.advs;
  scatter x=avisitn y=chg;
run;
""")
HTML(c['LST'])
```

Repeat the creation of the plot once more using Python. In this example, simple code from the Seaborn package was utilized to create the scatter plot.

```
In [3]: import seaborn as sns
sns.regplot(x=advs_sas2a["AVISITN"], y=advs_sas2a["CHG"])
sns.plt.show()
```

And finally, re-create the scatter plot for Rpy2, using the lattice package.

```
In [4]: import rpy2
import rpy2.rinterface as rinterface
import rpy2.robjects as robjects
from rpy2.robjects.packages import importr
from rpy2.robjects import Formula
from rpy2.robjects.vectors import IntVector, FloatVector
from rpy2.robjects.lib import grid

In [5]: rinterface.initr()
base = importr('base')
sas7bdat = importr('sas7bdat')
rprint = robjects.globalenv.get("print")
stats = importr('stats')
grdevices = importr('grDevices')
lattice = importr('lattice')

In [6]: robjects.r('advs_r <-
read.sas7bdat("C://Users//xxxx//studyXYZ//advs.sas7bdat")')
robjects.r('advs_r2 <- advs_r[advs_r$BASETYPE == "PHASE A",]')
robjects.r('advs_r3 <- advs_r2[advs_r2$AVISIT != "EARLY1",]')
r4 = robjects.r('advs_r4 <- advs_r3[advs_r3$AVISIT != "EARLY2",]')
xyplot = lattice.xyplot

formula = Formula('CHG ~ AVISITN')
formula.getenvironment()['AVISITN'] = r4.rx2('AVISITN')
formula.getenvironment()['CHG'] = r4.rx2('CHG')

p = lattice.xyplot(formula)
rprint(p)
```

By visual inspection of the three plots shown in the next page, you can compare and check that they match. If the task of plot creation and QC was split amongst two or more people using different programming languages, the SASPy and Rpy2 packages would prove useful for joining the codes back into a single file. It is possible to perform comparisons between other statistical packages, for example linear regression outputs (SAS: Proc GLM, Python: Statsmodels[5], R: lm). If the outputs can be standardized into a dataframe, then it is possible to perform comparisons using the function defined in the section above.

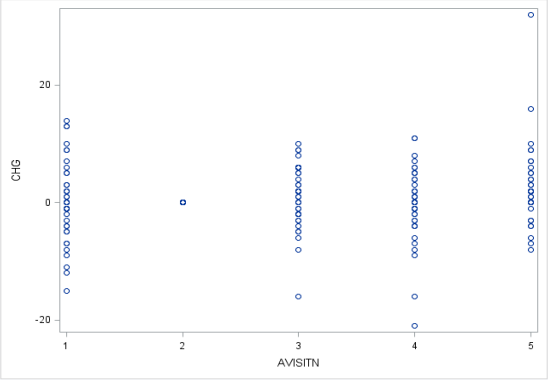

Figure 3. Plot using Proc SGplot (SASPy)

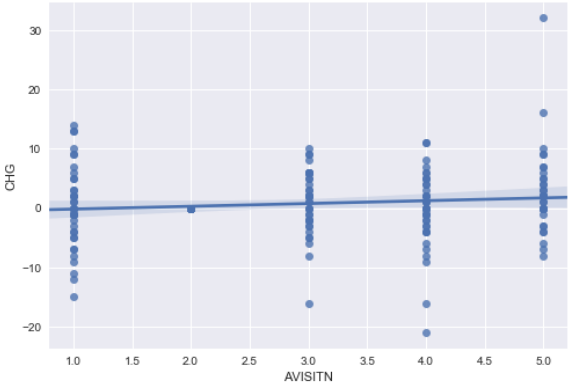

Figure 4. Plot using Seaborn (Python)

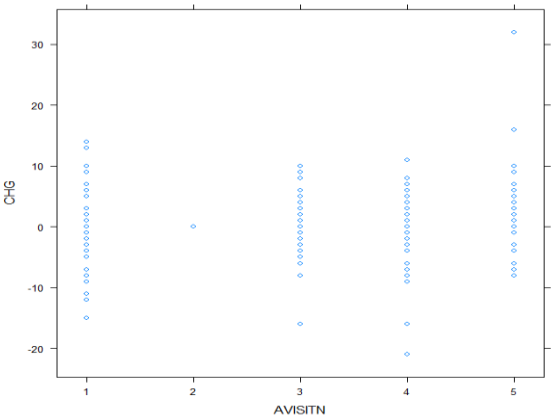

Figure 5. Plot using lattice (Rpy2)

PERSPECTIVE

It is thus possible to piece together code written by people from different job functions. A programmer could perform the necessary data manipulation (or SDTM/ADaM dataset creation) using SAS, a biostatistician analyses the datasets using packages from R, and a data visualisation analyst could create plots using Python, from end to end, all in a single program file without breaking the trail. Perhaps you had some data manipulation done using SAS but would like to utilize some machine learning techniques from Python's sk-learn package or R's xgboost package. Or you had a colleague that created some code using R, and you should continue where he left off using Python, to connect to the next part of the work process. Or your colleague is more comfortable using R while you are with SAS. Or the other way around.

We are no longer bound by the different programming languages. The possibilities of mixing and matching are truly endless.

CONCLUSION

This paper has suggested a method for independent program validations utilising the 3 programming languages SAS, R and Python within a single Python program. By converting outputs to dataframes, the function introduced in this paper allows users to perform comparisons at decimal places of their own choice. Validation continues to be an important aspect of statistical programming and as such, performing the validation accurately and efficiently will be beneficial to any organisation.

REFERENCES

1. SAS Institute. “SASPy”. Accessed January 26, 2018. Available at https://sassoftware.github.io/saspy/getting-started.html
2. Gautier, Laurent and rpy2 contributers. “Introduction to rpy2”. Accessed January 26, 2018. Available at http://rpy2.readthedocs.io/en/version_2.8.x/introduction.html
3. Les Shay. “Encoding and Decoding Strings (in Python 3.x)”. Accessed January 26, 2018. Available at http://pythoncentral.io/encoding-and-decoding-strings-in-python-3-x/
4. Watson, Richann and Johnson, Patty. 2011. “Automated or Manual Validation: Which One is for You?”. *Proceedings of the 2011 PharmaSUG Conference*. Available at https://www.pharmasug.org/proceedings/2011/AD/PharmaSUG-2011-AD01.pdf
5. Seabold, Skipper, and Josef Perktold. 2010. “Statsmodels: Econometric and statistical modeling with python.” *Proceedings of the 9th Python in Science Conference 2010*. Available at http://conference.scipy.org/proceedings/scipy2010/pdfs/seabold.pdf

ACKNOWLEDGEMENTS

I would like to thank Peter Rasmussen and Yang Mingyue for reviewing my paper, Nguyen Duc Hoa when I had questions specific to Python as well as Andy Hayden from the Stack Overflow community for providing help in dataframe comparisons.

RECOMMENDED READING

Butner, Daniel and Graham, Brandon. “Effective Independent Validation: Tips to Improve the Independent Validation Process”. *Proceedings of the 2013 PharmaSUG Conference*. Available at https://www.pharmasug.org/proceedings/2013/TF/PharmaSUG-2013-TF23.pdf

Randy Betancourt. “Python For SAS Users”. Available at http://nbviewer.jupyter.org/github/RandyBetancourt/PythonForSASUsers/blob/master/Chapter%20-1%20--%20Python%20for%20SAS%20Users%20Chapters.ipynb

DISCLAIMER

The contents of this paper are the work of the authors and do not necessarily represent the opinions, recommendations or practices of H. Lundbeck A/S.

CONTACT INFORMATION

Your comments and questions are greatly valued and encouraged. For any possible errors, please kindly contact the author at:

Aik Hoe Seah
Lundbeck Singapore
101 Thomson Rd,
#14-05 United Square,
Singapore 307591
aikh@lundbeck.com