Paper SAS4384-2020

Using Python with Model Studio for SAS® Visual Data Mining and Machine Learning

Jagruti Kanjia and Dominique Latour, Jesse Luebbert, SAS Institute Inc.

ABSTRACT

There are many benefits to using Python with Model Studio. It enables you to use the SAS® Scripting Wrapper for Analytics Transfer (SWAT) package with the SAS Code node. It makes interacting with SAS® Viya® easier for nontraditional SAS programmers within Model Studio. It is used within a SAS Code node for deployable data preparation and machine learning (with autogenerated reporting). It enables you to use packages built on top of SWAT, such as the SAS Deep Learning Python (DLPy) package, for deep learning within a SAS Code node. It gives you the ability to call Python in an existing open source environment from Model Studio to authenticate and transfer data to and from native Python structures. It lets you use your preferred Python environment for either data preparation or model building and call it through a SAS Code node for use or assessment within a pipeline. It enables you to use Python with the Open Source Code node. And it provides native Python integration for data preparation and model building. This paper discusses all these benefits and presents examples to show how you can take full advantage of them.

INTRODUCTION

The paper discusses specific ways in which open source software is embraced within SAS Viya. Python, a popular open source scripting language, is used to illustrate how you can integrate and use open source technology within Model Studio for SAS® Visual Data Mining and Machine Learning software.

Let's start by introducing the various software and components developed by SAS that the paper discusses.

SAS Viya

SAS Viya is a cloud-enabled, in-memory, distributed analytics extension of the SAS platform that makes it more scalable, fault-tolerant, and open. Open source integration enables analytical teams with varied backgrounds and experiences to come together and solve complex problems in new ways. One of the ways SAS embraces this openness is through Model Studio.

Model Studio

Model Studio is a node-based, drag-and-drop graphical user interface for SAS Viya that is designed to accelerate time to value for building analytics pipelines. The pipelines can be machine learning, forecasting, or text analytics pipelines. In this paper, the focus is on the SAS Visual Data Mining and Machine Learning component of Model Studio. Model Studio provides you with various techniques for data preparation, model building, model ensembling, and model comparison. The advanced template for a categorical target shown in Figure 1 is an example of a Model Studio pipeline.

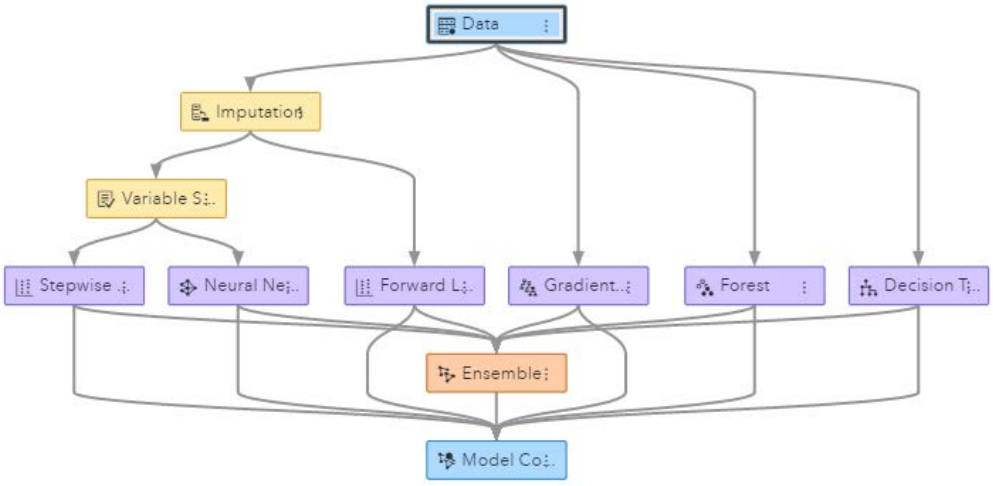

Figure 1. Example of a Model Studio Pipeline

Model Studio is also designed to be extensible through nodes that are available in the Miscellaneous node category. The paper focuses on two of those nodes: the SAS Code node and the Open Source Code node.

SAS Code Node and Open Source Code Node

You can use the SAS Code node within Model Studio for data preparation and model building. This node enables you to add custom functionality to the software. This functionality can be traditional SAS code and procedures, SAS Viya procedures that call SAS® Cloud Analytic Services (CAS) actions, or even open source code. You can use the SWAT package within the SAS Code node. The Open Source Code node is a Miscellaneous node that can run Python or R code. The node can also be treated as a Supervised Learning node so that your Python or R model can be assessed and compared with other Model Studio models.

SWAT

SAS Scripting Wrapper for Analytics Transfer (SWAT) is an open source Python package that makes it easy for programmers who are familiar with Python syntax to use the distributed, in-memory code base of SAS Viya and CAS for Python 2.7.x or 3.4 and later.

USING PYTHON IN THE OPEN SOURCE CODE NODE

The Open Source Code node can execute Python or R scripts that are on the same machine as the compute server for SAS Viya in order to do data visualization, data preparation, or model building. This paper focuses on Python scripts.

DATA PREPARATION USING PYTHON IN AN OPEN SOURCE CODE NODE

It is possible to do data preparation by using native Python syntax and have the output data frame from the script be passed on to subsequent nodes. However, this is not something

that can be deployed; it is designed for prototyping various feature engineering techniques. In order to pass the output data to the next node, you need to select the Use output data in child nodes property. The following Python code shows how to create a new feature, MORTPAID (the amount of the mortgage paid), simply by subtracting the value of the mortgage due (MORTDUE) from the total mortgage value (VALUE), using Pandas DataFrames that subsequent nodes will use.

The dm_inputdf and dm_scoreddf data items are set by the Model Studio environment and correspond to Pandas DataFrames for the sampled input data and scored input data, respectively. You can find a complete list of generated data items in the Appendix.

```
# Create a new variable to use in Python
dm_inputdf['MORTPAID'] = dm_inputdf['VALUE'] - dm_inputdf['MORTDUE']
# Point output data name to modified data set
dm_scoreddf = dm_inputdf
```

MODEL BUILDING USING PYTHON IN AN OPEN SOURCE CODE NODE

It is possible to do open source modeling by using various open source packages. One popular package for supervised machine learning is LightGBM. "Light" refers to its high speed; GBM stands for gradient boosting machine, the algorithm that this method uses. The LightGBM Python package is a gradient boosting framework that uses a tree-based learning algorithm. LightGBM grows trees vertically, whereas other boosting algorithms grow trees horizontally; this means that LightGBM grows tree leafwise, and other algorithms grow trees levelwise. It chooses the leaf with maximum delta loss to grow. A loss function is a measure that quantifies how well the prediction model is able to predict the expected outcome. When growing the same leaf, a leafwise algorithm can reduce more loss than a levelwise algorithm can. Because of its high speed, LightGBM can also process large data tables while using less memory than other boosting algorithms.

Invoking the LightGBM package is easy. What is difficult is finding a good set of parameters—that is, parameter tuning. LightGBM supports more than 100 parameters. Table 1 lists some of the basic parameters that you can use to control this algorithm.

Table 1. Basic Parameters Available in LightGBM Algorithm

LightGBM Parameter	Description
bagging fraction	Specifies the fraction of data to be used for each iteration. This parameter is generally used to speed up the training and avoid overfitting.
bagging_freq	Specifies the frequency of bagging. The value of 0 disables bagging; the value indicates to perform bagging at every *k*th iteration.
boosting_type	Specifies the type of algorithm that you want to run. The *gbdt* is a traditional gradient boosting decision tree algorithm. Other supported values are *rf* or *random_forest* (random forest), *dart* (dropouts in multiple additive regression tree), and *goss* (gradient-based one-side sampling).

LightGBM Parameter	Description
feature_fraction	Specifies the percentage of features randomly selected at each iteration of building trees. This parameter is used when the boosting type is set to *random forest*. For example, a 0.75 feature fraction means that LightGBM selects 75% of the parameters randomly at each iteration.
learning_rate	Specifies the learning rate parameter. This value determines the impact of each tree on the final outcome. A GBM algorithm works by starting with an initial estimate that gets updated using the output of each tree. The learning parameter controls the magnitude of this change in the estimates. Typical values: 0.1, 0.001, 0.003, etc.
metric	Specifies the metric to be evaluated in the evaluation set, such as gamma, area under the curve, binary log-loss, or binary error
min_data_per_group	Specifies the minimal amount of data per categorical group. The default is 100.
num_iterations	Specifies the number of boosting iterations
num_leaves	Specifies the number of leaves in one tree. The default is 31.
objective or objective_type	Specifies the application of the model, whether it is a regression problem or classification problem. By default, LightGBM treats the model as a regression model. This is probably the most important parameter. Specify the value of *binary* for binary classification.

In order to use the LightGBM package, you must first convert the training and validation data into LightGBM data sets. After creating the lgb_train and lgb_valid data sets, a Python dictionary is created that contains a set of parameters and associated values. Because this is a classification problem, "binary" is used as the objective type and "binary_logloss" as the metric. The boosting type is set to "gbdt" since you are implementing the gradient boosting algorithm. The following methods are used to produce some results:

- plot_importance plots the model feature importance.
- plot_tree plots a specified tree.
- plot_split_value_histogram generates a split-value histogram for the specified feature of the model.

Now let's look at an example of how you can use this algorithm in the Open Source Code node. The new pipeline shown in Figure 2 is created in a Model Studio project. The Open Source Code node is preceded by an Imputation node to handle the missing values that occur in some of the variables.

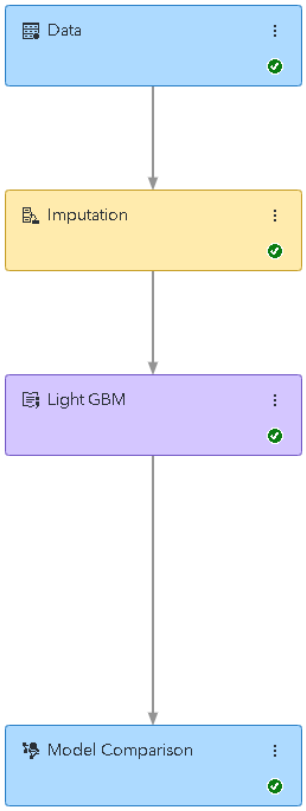

Figure 2. Model Studio Pipeline with Imputation Node and Open Source Code Node

In the Open Source Code node, you enter the following code in the Code editor. The code takes advantage of data items that are set by the Model Studio environment to integrate the Open Source Code node with the pipeline. The dm_inputdf data item identifies the input Pandas DataFrame, whereas the dm_partitionvar and dm_partition_train_val data items identify the partition variable and partition value, respectively, that correspond to the training observations. Model Studio supports a single partition variable whose values identify the training, validation, and test observations, so you need to create light GBM training and validation data sets. Note that currently the Open Source Code node does not create data items for validation or test observations. When Model Studio generates a partition variable, the values 1, 0, and 2 are assigned to the observations of the training, validation, and test partitions, respectively. This is why the following code uses the expression `dm_inputdf[dm_partitionvar] == 0` to produce the validation data set:

```
import lightgbm as lgb
import os
import sys
import matplotlib as mpl
if os.environ.get('DISPLAY','') == '':
    print('no display found. Using noninteractive Agg back end')
    mpl.use('Agg')
import matplotlib.pyplot as plt

# Make sure nominals are category
dtypes = dm_inputdf.dtypes
nominals = dtypes[dtypes=='object'].keys().tolist()
```

```
for col in nominals:
    dm_inputdf[col] = dm_inputdf[col].astype('category')

# Training set
train = dm_inputdf[dm_inputdf[dm_partitionvar] == dm_partition_train_val]
X_train = train.loc[:,dm_input]
y_train = train[dm_dec_target]
lgb_train = lgb.Dataset(X_train, y_train, free_raw_data = False)

# Validation set for early stopping (optional)
valid = dm_inputdf[dm_inputdf[dm_partitionvar] == 0]
X_valid = valid.loc[:,dm_input]
y_valid = valid[dm_dec_target]
lgb_valid = lgb.Dataset(X_valid, y_valid, free_raw_data = False)

# LightGBM parameters
params = {
    'num_iterations': 60,
    'boosting_type': 'gbdt',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'num_leaves': 75,
    'learning_rate': 0.05,
    'feature_fraction': 0.75,
    'bagging_fraction': 0.75,
    'bagging_freq': 0,
    'min_data_per_group': 10
}

evals_result = {}  # to record eval results for plotting

# Fit LightGBM model to training data
gbm = lgb.train(
    params,
    lgb_train,
    valid_sets = [lgb_valid, lgb_train],
    valid_names = ['valid','train'],
    early_stopping_rounds = 5,
    evals_result=evals_result
)

ax = lgb.plot_tree(gbm, tree_index=53, figsize=(25, 15),
show_info=['split_gain'])
plt.savefig(dm_nodedir + '/rpt_tree.png', dpi=500)

print('Plotting feature importances...')
ax = lgb.plot_importance(gbm, max_num_features=10)
plt.savefig(dm_nodedir + '/rpt_importance.png', pad_inches=0.1)

print('Plotting split-value histogram...')
ax = lgb.plot_split_value_histogram(gbm, feature='IMP_CLNO', bins='auto')
plt.savefig(dm_nodedir + '/rpt_hist1.png')

# Generate predictions and create new columns for Model Studio
tmp = gbm.predict(dm_inputdf.loc[:,dm_input])
dm_scoreddf = pd.DataFrame()
dm_scoreddf[dm_predictionvar[1]] = tmp
dm_scoreddf[dm_predictionvar[0]] = 1 - tmp
```

After you save the code, the pipeline is run. The Open Source Code node executes Python code in the Python environment that is installed on the compute server where SAS Viya is running. The Node tab in Figure 3 displays the generated plot: the specified tree, the model feature importance plot, and the split-value histogram for the imputed variable CLNO.

In its Results window, the Open Source Code node can display plots, tables, and files that the Python code generates, provided that those files are created in the node directory, which is specified by the dm_nodedir data item. The files must have the rpt_ prefix and one of the following extensions: .jpg (plots and images), .csv (tables), or .txt (text files).

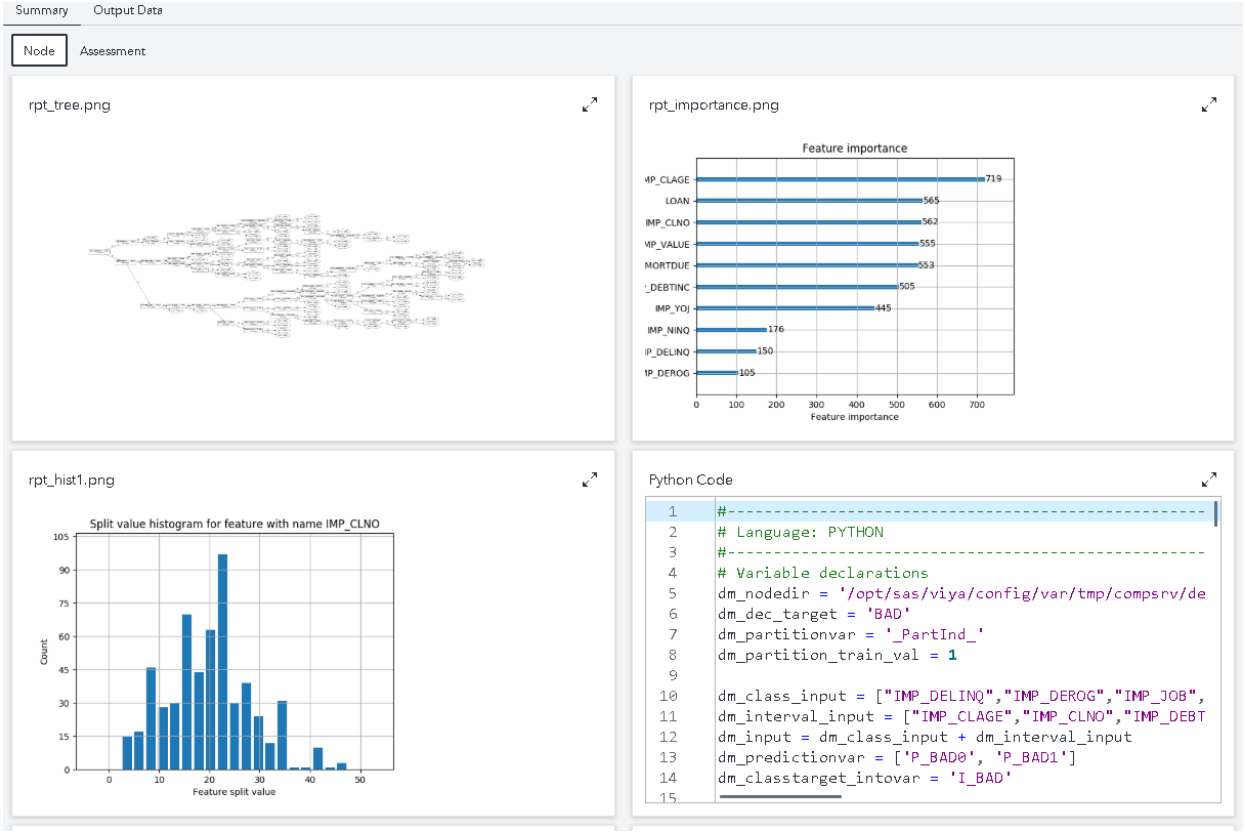

Figure 3. Open Source Code Node Results

The assessment results are calculated from the scored sampled project table that corresponds to the dm_scoreddf data item. The Assessment tab in Figure 4 displays the assessment reports: lift reports, ROC reports, fit statistics table, and event classification chart. The fit statistics table shows that the misclassification rate is 0.0274 for the training partition, 0.0928 for the validation partition, and 0.0923 for the test partition.

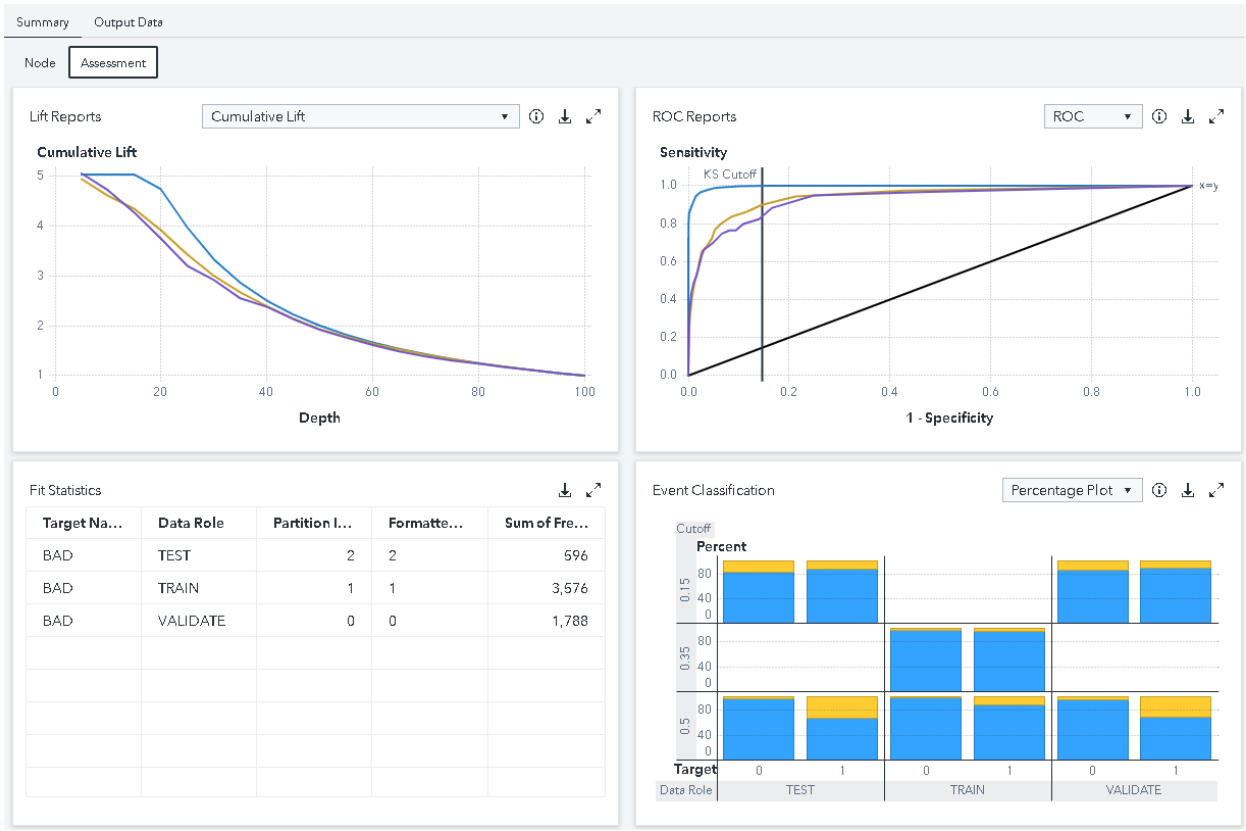

Figure 4. Open Source Code Node Assessment Results

USING SWAT IN THE SAS CODE NODE

Because you can use the SWAT package in a SAS Code node, a user familiar with Python programming can extend the functionality of Model Studio. You can use the Base SAS® Java Object to execute a Python script. Of note, a model that is written using SWAT can be deployable, given that it generates either DATA step (DS1) score code or a SAS analytic store (astore). Now let's look at some examples of how you can produce deployable assets for both data preparation and model building.

DEPLOYABLE DATA PREPARATION USING SWAT IN A SAS CODE NODE

You can execute the following code in a SAS Code node as a data mining preprocessing node—that is, as an Unsupervised Learning node. The dmcas_varmacro macro is used here to generate a comma-separated list of categorical variable names enclosed in single quotation marks.

```
/* Create additional variable lists */
%dmcas_varmacro(name=dm_class_input, metadata=&dm_metadata,
where=%nrbquote(level in ('NOMINAL','ORDINAL','BINARY') and role = 'INPUT'),
key=NAME, quote=Y, singlequote=Y, comma=Y, append=Y);
```

Then you create a Python script as follows that invokes the CAS transform action to perform a moment transformation of all the categorical variables. The CAS session macro variables that Model Studio initializes (dm_cashost, dm_casport, and dm_cassessionid) are used to

connect to the CAS session that is started when the SAS Code node runs. The dm_rstoretable macro variable identifies the SAS analytic store that can be used to score a table to generate the transformed variables.

```
/* Create Python script with SAS macro variables inserted */
proc cas;
   file _codef "&dm_nodedir&dm_dsep.dm_srcfile.py";
   print "import swat";
   print "s=swat.CAS(hostname='&dm_cashost', port='&dm_casport',
          session='&dm_cassessionid')";
   print "s.transform(table=dict(caslib='&dm_caslib', name='&dm_memname',
where='&dm_partitionvar=&dm_partition_train_val'),
requestPackages=dict(function=dict(name='TE', inputs=[%dm_class_input],
targets='&dm_dec_vvntarget', event='&dm_dec_event',
targetsinheritformats=True, inputsinheritformats=True,
mapInterval=dict(method='moments',  args=dict(includeMissingLevel=True,
nMoments=1)))), savestate=dict(caslib='&dm_caslib', name='&dm_rstoretable',
replace=True))";
run;
```

Next, you pass the Python executable file that you created earlier to the Java class SASJavaExec by using the Base SAS Java Object. SASJavaExec runs the Python script with its command line argument and passes any output or reports any errors back to the SAS Viya log. In the following DATA step, you initiate the Java Object by giving the class name (com.sas.analytics.datamining.servertier.SASPythonExec) and the Python script name. (You can find a description of the Java class SASJavaExec in the following white paper: https://github.com/sassoftware/enlighten-integration/blob/master/SAS_Base_OpenSrcIntegration/SAS_Base_OpenSrcIntegration.pdf.)

```
/* Set class path */
%dmcas_setClasspath();

/* Execute Python script */
data _null_;
    length rtn_val 8;
    declare javaobj
    j("com.sas.analytics.datamining.servertier.SASPythonExec",
      "&dm_nodedir&dm_dsep.dm_srcfile.py");
    j.callVoidMethod("setOutputFile",
                     "&dm_nodedir&dm_dsep&lang._output.txt");
    j.callIntMethod("executeProcess", rtn_val);
    j.delete();
    call symput('javaobj_rtnval', rtn_val);
run;
```

The following code shows how you use the dm_metadata macro variable that identifies the table containing the variables and attributes to reject the input categorical variables that were transformed. The dm_file_deltacode macro variable identifies a file that contains the specified metadata changes that successor nodes will use.

```
/* Reject original inputs (optional) */
filename deltac "&dm_file_deltacode";

data _null_;
     file deltac;
     set &dm_metadata;
```

```
     length codeline $ 500;
     if level in ('NOMINAL','ORDINAL','BINARY') and role = 'INPUT' then do;
          codeline = "if upcase(NAME) = '"!!upcase(tranwrd(ktrim(NAME), "'",
"''"))!!"' then do;";
          put codeline;

          codeline = "ROLE='REJECTED';";
          put +3 codeline;
          put 'end;';
          output;
     end;
run;

filename deltac;
```

DEPLOYABLE MODEL BUILDING USING SWAT IN A SAS CODE NODE

You can execute the following code in a SAS Code node that has been moved into the Supervised Learning lane of Model Studio. This code produces a gradient boosting tree model by calling the gbtrain action of the decisionTree action set.

```
/* Create additional variable lists */
%dmcas_varmacro(name=dm_class_var, metadata=&dm_metadata,
where=%nrbquote(level in ('NOMINAL','ORDINAL','BINARY') and role in
('INPUT','TARGET')), key=NAME, quote=Y, singlequote=Y, comma=Y, append=Y);
%dmcas_varmacro(name=dm_input, metadata=&dm_metadata,
where=%nrbquote(role='INPUT'), key=NAME, quote=Y, singlequote=Y, comma=Y,
append=Y);

* Create Python script with SAS macro variables inserted;
proc cas;
    file _codef "&dm_nodedir&dm_dsep.dm_srcfile.py";
    print "import swat";
    print "s=swat.CAS(hostname='&dm_cashost', port='&dm_casport',
session='&dm_cassessionid')";
    print "s.loadactionset('decisiontree')";
    print "s.gbtreetrain(table=dict(caslib='&dm_caslib', name='&dm_memname',
where='&dm_partitionvar=&dm_partition_train_val'),
target='&dm_dec_vvntarget', inputs=[%dm_input], nominals=[%dm_class_var],
savestate=dict(caslib='&dm_caslib', name='&dm_rstoretable', replace=True))";
run;

/* Set class path */
%dmcas_setClasspath();

/* Execute Python script */
data _null_;
    length rtn_val 8;
    declare javaobj
j("com.sas.analytics.datamining.servertier.SASPythonExec",
"&dm_nodedir&dm_dsep.dm_srcfile.py");
    j.callVoidMethod("setOutputFile",
"&dm_nodedir&dm_dsep&lang._output.txt");
    j.callIntMethod("executeProcess", rtn_val);
    j.delete();
    call symput('javaobj_rtnval', rtn_val);
run;
```

DEPLOYABLE MODEL BUILDING USING DLPy IN A SAS CODE NODE

Several Python packages that use SWAT are tailored to specific use cases. One extremely popular use case is the SAS Deep Learning Python (DLPy) package. This package was designed specifically to make programming deep learning models more approachable by using friendly Keras-like APIs. Conveniently enough, this package can be used within a SAS Code node as well.

In this example, the SAS Code node is preceded by the Feature Machine node, which is a data mining preprocessing node that generates features that address one or more transformation policies. The new features can be generated to fix data quality issues such as high cardinality, high kurtosis, high skewness, low entropy, outliers, and missing values.

The new pipeline shown in Figure 5 is created in a Model Studio project.

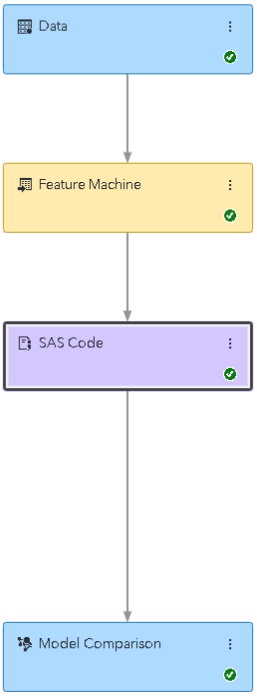

Figure 5. Model Studio Pipeline with Feature Machine and SAS Code Nodes

The following code is specified in the Training code editor of the SAS Code node. Two lists of variables are generated using the dmcas_varmacro macro: one for categorical input variables and one for all inputs, interval and categorical. The DLPy package uses the SAS Viya deep neural network action set deepLearn to generate the model. When the node is run, the dm_srcfile.py file is executed in the Python environment installed on the compute server that is running SAS Viya.

```
/* Create additional variable lists */

%dmcas_varmacro(name=dm_class_var, metadata=&dm_metadata,
where=%nrbquote(level in ('NOMINAL','ORDINAL','BINARY') and role in
('INPUT','TARGET')), key=NAME, quote=Y, singlequote=Y, comma=Y, append=Y);
%dmcas_varmacro(name=dm_input, metadata=&dm_metadata, where=%nrbquote(role =
'INPUT'), key=NAME, quote=Y, singlequote=Y, comma=Y, append=Y);
```

```
/* Create Python file to execute */
proc cas;
    file _codef "&dm_nodedir&dm_dsep.dm_srcfile.py";
    print "import swat";
    print "from dlpy import Model, Sequential";
    print "from dlpy.model import Optimizer, AdamSolver";
    print "from dlpy.layers import *";
    print "import pandas as pd";
    print "import os";
    print "import matplotlib";
    print "from matplotlib import pyplot as plt";
    print "plt.switch_backend('agg')";
    print "s = swat.CAS(hostname='&dm_cashost', port='&dm_casport',
session='&dm_cassessionid')";
    print "s.loadactionset('deeplearn')";
    print "s.setsessopt(caslib='&dm_caslib')";
    print "model = Sequential(s,
model_table=s.CASTable('simple_dnn_classifier', replace=True))";
    print "model.add(InputLayer(std='STD'))";
    print "model.add(Dense(20, act='relu'))";
    print "model.add(OutputLayer(act='softmax', n=2, error='entropy'))";
    print "model.fit(s.CASTable('&dm_memname',
where='&dm_partitionvar=&dm_partition_train_val'),
target='&dm_dec_vvntarget', inputs=[%dm_input], nominals=[%dm_class_var],
optimizer=Optimizer(algorithm=AdamSolver(learning_rate=0.005,learning_rate_po
licy='step',gamma=0.9,step_size=5), mini_batch_size=4, seed=1234,
max_epochs=50))";
```

In the following code, the Graphviz utility is used to create and save plots as image files. The directed acyclic graph (DAG) of the model network is displayed. The summary of model training history was written to the _output.txt file, which is displayed as Python output. A plot is created to visualize the training history and saved in an image file.

```
    print "outF = open('&dm_nodedir/_output.txt', 'w')";
    print "summary = model.print_summary()";
    print "print(summary, sep=' ', end='\n\n', file=outF, flush=False)";
    print "history = model.training_history";
    print "print(history, sep=' ', end='\n\n', file=outF, flush=False)";
    print "n=model.plot_network()";
    print "from graphviz import Graph";
    print "g = Graph(format='png')";
    print "n.format = 'png'";
    print "n.render('&dm_nodedir&dm_dsep.rpt_network1.gv')";
    print "outF.close()";
    print "th=model.plot_training_history(fig_size=(15,6))";
    print
"th.get_figure().savefig('&dm_nodedir&dm_dsep.rpt_train_hist.png')";
```

The dlExportModel action from the Deep Neural action set is used as follows to create the analytic store table named by the dm_rstoretable macro variable. When the script finishes running, Model Studio uses this analytic store to score the entire training table so that assessment results can be produced.

```
    print "s.dlExportModel(modeltable='simple_dnn_classifier',
initWeights='simple_dnn_classifier_weights', randomflip='NONE',
randomCrop='NONE', randomMutation='NONE', casout='&dm_rstoretable')";
run;
```

```
/* Set class path */
%dmcas_setClasspath();
/* Execute Python script */

data _null_;
    length rtn_val 8;
    declare javaobj
j("com.sas.analytics.datamining.servertier.SASPythonExec",
"&dm_nodedir&dm_dsep.dm_srcfile.py");
    j.callVoidMethod("setOutputFile",
"&dm_nodedir&dm_dsep&lang._output.txt");
    j.callIntMethod("executeProcess", rtn_val);
    j.delete();
    call symput('javaobj_rtnval', rtn_val);
run;

%let source=&dm_nodedir&dm_dsep.;

/* Rename png file that was created using graphviz function */
data _null_;
   rc=rename("&dm_nodedir&dm_dsep.rpt_network1.gv.png",
"&dm_nodedir&dm_dsep.rpt_network1_gv.png", "file");
run;
```

In contrast with the Open Source Code node, which automatically generates reports, you have to use the dmcas_report utility macro as follows to indicate which reports to display in addition to the reports that Model Studio automatically generates:

```
/* Create a report to display network plot */
%dmcas_report(file=&dm_nodedir&dm_dsep.rpt_network1_gv.png, reportType=Image,
description=%nrbquote('Network Plot'), localize=N);

/* Create a report to display training history plot */
%dmcas_report(file=&dm_nodedir&dm_dsep.rpt_train_hist.png, reportType=Image,
description=%nrbquote('Tree Plot'), localize=N);

/* Create a report to display Python output file */
%dmcas_report(file=&dm_nodedir&dm_dsep._output.txt, reportType=CodeEditor,
type=TEXT, description=%nrbquote('Python Output'), localize=N);
```

The results of the SAS Code node are displayed in Figure 6. This includes the submitted SAS Code, an automatic report produced by Model Studio, and the three requested reports: two image files and the text file containing the Python output.

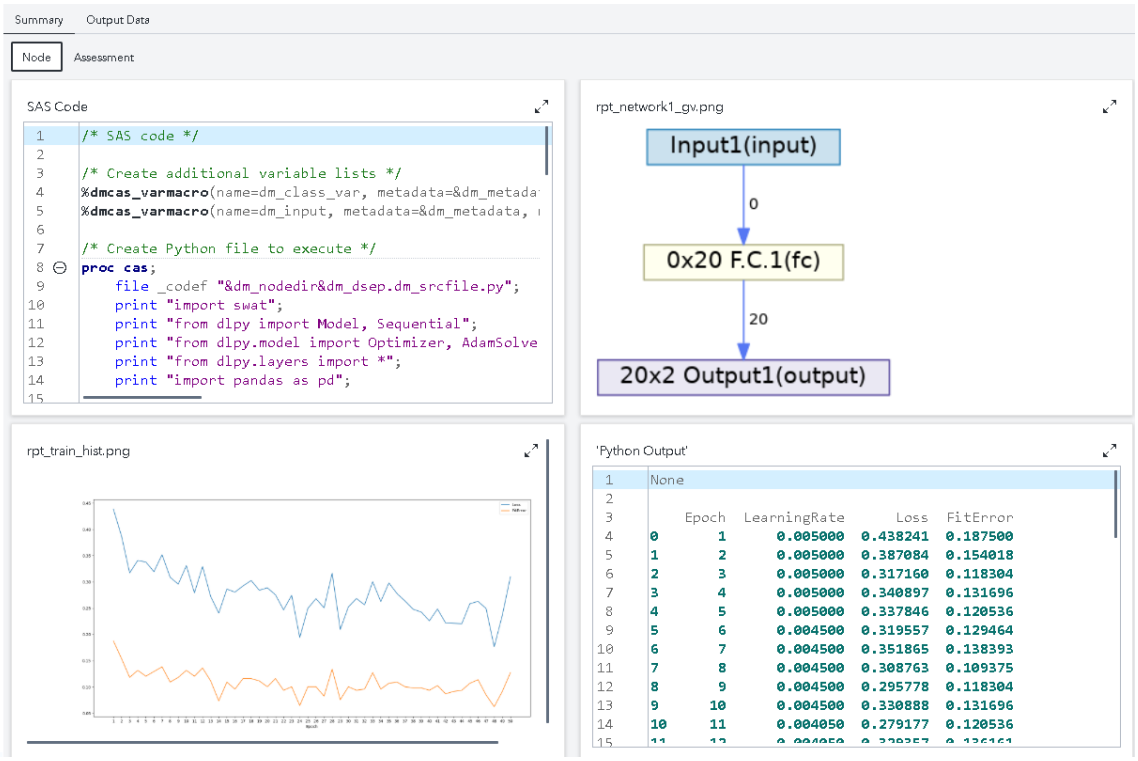

Figure 6. SAS Code Node Results

The Assessment tab in Figure 7 displays the assessment reports: lift reports, ROC reports, fit statistics table, and event classification chart. The fit statistics table shows that the misclassification rate is 0.0903 for the training partition, 0.1079 for the validation partition, and 0.1040 for the test partition.

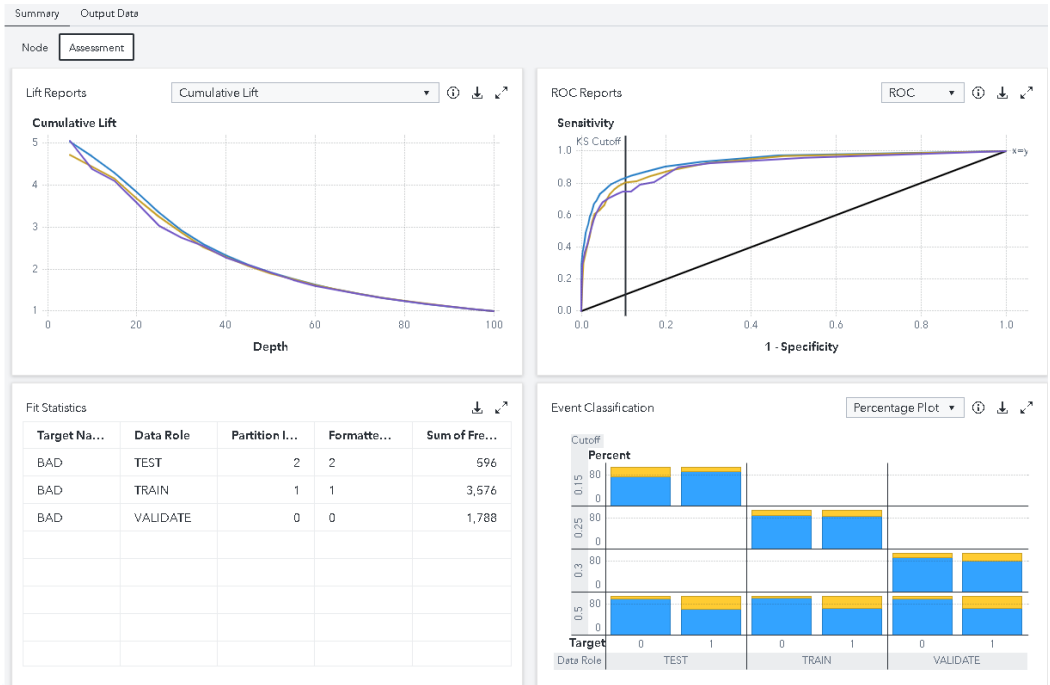

Figure 7. SAS Code Assessment Results

Model Studio supports a variety of model interpretability reports, such as partial dependence (PD) plots and individual conditional expectation (ICE) plots. Figure 8 shows all supported global interpretability and local interpretability reports that you can create. The pipeline is rerun to create these reports. The figure displays the surrogate model variable importance table, PD plot, PD and ICE overlay plot, and LIME explanations plot on the Model Interpretability tab.

Note that the generation of such reports is possible only because there is a representation of the model as score code; in this case it is an analytic store. If the model does not produce score code but instead generates only a scored table, then the only model interpretability report that you can generate is the surrogate variable importance plot, because the other reports require rescoring.

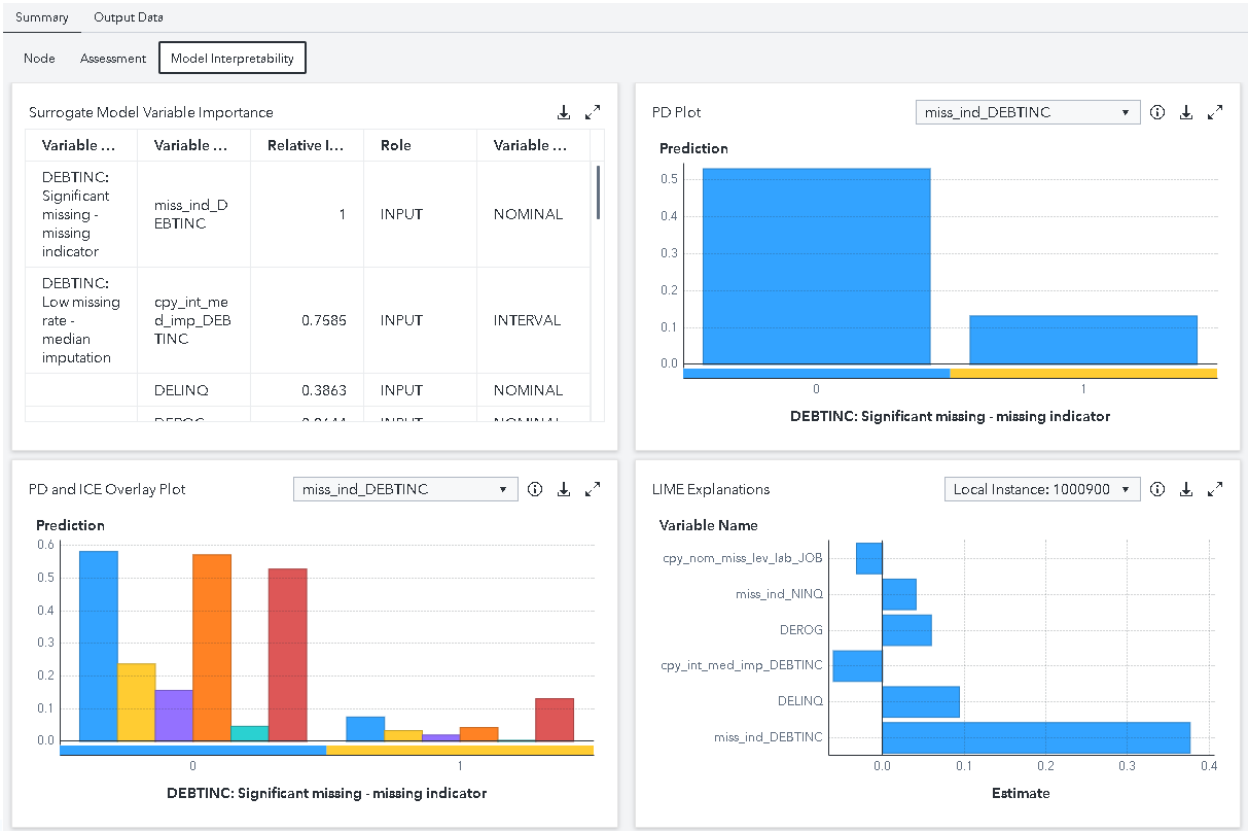

Figure 8. SAS Code Node Model Interpretability Results

USING PYTHON IN AN EXISTING OPEN SOURCE ENVIRONMENT FROM MODEL STUDIO TO AUTHENTICATE AND TRANSFER DATA TO AND FROM NATIVE PYTHON STRUCTURES

As explained earlier, the Open Source Code node executes Python scripts that are on the same machine as the compute server where SAS Viya is running. However, you can submit Python scripts on a remote server by using the remrunner and Paramiko packages.

The remrunner (remote runner) package enables you to transfer a local script file to a remote host and execute it. The named file is copied to a temporary location on a remote host, its permissions are set to 0700, and the script is then executed. During cleanup, the

[PID] directory and all its contents are removed before the connections are closed. The remrunner package uses the Paramiko package for SSHv2 protocol implementation. The current limitation of the remrunner package is that it assumes that SSH keys, which allow password-free log-ins, are already in place. There is no option to prompt for a password or SSH passphrase.

In this example, the SAS Code node is used to create and submit a Python file to a remote system and transfer some of the results back to the system where SAS Viya is running. The Feature Machine node is used to generate new features by performing variable transformations to improve data quality and model accuracy.

The SAS Code node is again used as the Supervised Learning node. However, this time it creates and submits a Python file to a remote system and displays results from executing the Python model. This differs from the previous examples that use the Open Source Code node, where a sample of the data was downloaded to the SAS Viya client and then converted to a Pandas DataFrame. Here SWAT is used to directly access the training table loaded into CAS and to create a DataFrame from the entire table. Then the scored table that is produced by an XGB Classifier Python model is uploaded into the CAS session so that Model Studio can assess the entire score table for this Python-generated model.

The new pipeline shown in Figure 9 is created in Model Studio.

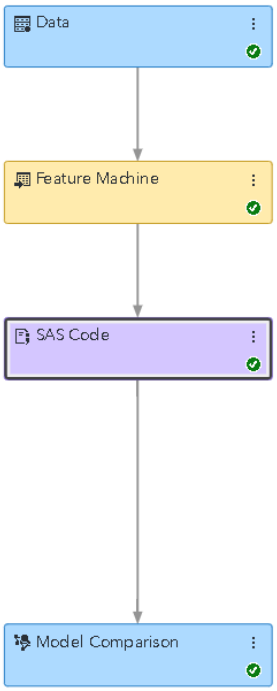

Figure 9. Model Studio Pipeline with Feature Machine and SAS Code Nodes

You can find the complete code that is specified in the SAS Code node editor in GitHub at the following link: https://github.com/sascommunities/sas-global-forum-2020/tree/master/papers/4384-2020-Kanjia

As before, you start by creating two variable lists: one for categorical input variables, and the other for all inputs, interval and categorical.

```
* Create additional variable lists;
%dmcas_varmacro(name=dm_class_var, metadata=&dm_metadata,
where=%nrbquote(level in ('NOMINAL','ORDINAL','BINARY') and role in
('INPUT','TARGET')), key=NAME, quote=Y, singlequote=Y, comma=Y, append=Y);
%dmcas_varmacro(name=dm_input, metadata=&dm_metadata, where=%nrbquote(role =
'INPUT'), key=NAME, quote=Y, singlequote=Y, comma=Y, append=Y);
```

Then you use the first CAS procedure step to create a Python file named gb01.py, which when executed runs an XGB classifier algorithm. This algorithm is an implementation of gradient boosting decision trees designed for speed and performance.

```
proc cas;
      file _codef "&dm_nodedir&dm_dsep.gb01.py";
      print "#!/usr/bin/python";
      print "import os";
      print "import paramiko";
      print "import pandas as pd";
      print "import swat, sys";
      print "from sklearn import ensemble";
      print "from xgboost import XGBClassifier";
      print "from xgboost import plot_tree";
      print "import matplotlib";
      print "matplotlib.use('Agg')";
      print "from matplotlib import pyplot as plt";
      print "os.environ['CAS_CLIENT_SSL_CA_LIST'] =
'/tmp/jk/mon_trustedcerts.pem'";
      print "conn = swat.CAS(hostname = '&dm_cashost', port = '&dm_casport',
session = '&dm_cassessionid', authinfo='~/.authinfo')";
      print "table = '&dm_memname'";
      print "nodeid = '&dm_nodeid'";
      print "caslib = '&dm_caslib'";
      print "dm_partitionvar = '&dm_partitionvar'";
      print "dm_inputdf = conn.CASTable(caslib = caslib, name =
table).to_frame()";
      print "dm_traindf = dm_inputdf[dm_inputdf[dm_partitionvar] == 1]";
      print "outF = open('_output.txt', 'w')";
      print "print(dm_traindf.head(), sep=' ', end='\n\n', file=outF,
flush=False)";
      print "X_train = dm_traindf.loc[:,[%dm_input]]";
      print "y_train = dm_traindf['%qktrim(&dm_dec_vvntarget)']";
      print "X = dm_inputdf.loc[:,[%dm_input]]";
      print "xgb = XGBClassifier()";
      print "xgb.fit(X_train, y_train)";
      print "print(xgb, sep=' ', end='\n\n', file=outF, flush=False)";
      print "pred = xgb.predict_proba(X)";
      print "dm_inputdf['%qktrim(&dm_predicted_vvnvar)'] = pred[:,1]";
      print "dm_inputdf['P_BAD0'] = pred[:,0]";
      print "dm_inputdf['%qktrim(&dm_into_vvnvar)'] =
pd.DataFrame(xgb.predict(X))";
      print "print(dm_inputdf.head(), sep=' ', end='\n', file=outF,
flush=False)";
      print "outF.close()";
      print "plot_tree(xgb,num_trees=3)";
      print "fig = plt.gcf()";
      print "fig.set_size_inches(15,10)";
      print "fig.savefig('rpt_tree.png',dpi=700)";
```

```
      print "conn.upload_frame(dm_inputdf, casout = dict(name = nodeid +
'_score', caslib = caslib, replace = True))";
      print "varimp = pd.DataFrame(list(zip(X_train,
xgb.feature_importances_)), columns=['Variable Name', 'Importance'])";
      print "conn.upload_frame(varimp, casout = dict(name = 'gb_varimp',
caslib = caslib, replace = True))";
      print "ssh_client=paramiko.SSHClient()";
      print
"ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())";
      print "ssh_client.connect(hostname='python.emd.sashq-
d.openstack.sas.com', username='&sysuserid')";
      print "ftp_client=ssh_client.open_sftp()";
      print
"ftp_client.put('rpt_tree.png','&dm_nodedir&dm_dsep.rpt_tree.png')";
      print
"ftp_client.put('_output.txt','&dm_nodedir&dm_dsep._output.txt')";
 run;
```

The first statement in the script, which follows, indicates that the content of the file will be interpreted by the Python binary that is located at /usr/bin/python. This enables your script to run in a Python environment no matter where Python is installed.

```
print "#!/usr/bin/python";
```

Then you import the required packages. Here Pandas is used to create DataFrames; SWAT is used to create a connection to the CAS server and upload the data; xgboost is used to build the XGBoost model; matplotlib is used to create plots; and the os package is used to set the path to the certificate in the CAS_CLIENT_SSL_CA_LIST environment variable in the environment where you want to run Python.

If your CAS server is configured to use SSL for communication, you must configure your certificate on the client side. Note that beginning in SAS Viya 3.3, encrypted communication is enabled in the server by default.

When your CAS server is configured with encryption, if you try to connect to a CAS server from Python by using SWAT, you get an error message like "Error: The TCP/IP negClientSSL support routine failed with status 807ff013.\n." You can avoid this error by setting the path to the correct certificate in the CAS_CLIENT_SSL_CA_LIST environment variable in the environment where you are running Python, as follows. The path shown here is a client-side path, so the certificates are typically copied to a local directory from the server.

```
CAS_CLIENT_SSL_CA_LIST='/path/to/certificates.pem'
```

Next, you set the path to the correct certificate in the CAS_CLIENT_SSL_CA_LIST environment variable in the environment where you are running Python:

```
os.environ['CAS_CLIENT_SSL_CA_LIST'] = /tmp/viya_trustedcerts.pem'
```

To enable a Python program to work with CAS, you must establish a connection with the CAS server, as shown in the following code. You need a hostname and a port that the CAS controller is listening on. You must have an authinfo file so that you can specify your credentials to the controller. SWAT is the name of the package that you use to connect to this server. It is assumed that a CAS server is running. The SWAT package can connect to

either the binary port or the HTTP port. If you have the option of using either, the binary port provides better performance. To connect to a CAS server, you simply import SWAT and use the swat.CAS class to create a connection. The dm_cashost and dm_casport macro variables are initialized to point to the CAS server that Model Studio uses. The dm_cassessionid macro variable resolves to the CAS session associated with the SAS Code node. It uses the authinfo file located in the user's home directory to authenticate your identity from a remote Python system.

```
# Use the swat.CAS class to create connection to CAS running on SAS Viya
system
conn = swat.CAS(hostname = '&dm_cashost', port = '&dm_casport', session =
'&dm_cassessionid', authinfo='~/.authinfo')
```

The easiest way to get data into CAS is to use the data loading methods on the CAS object that provide parallel data reading operations in the Pandas module. These operations include pandas.read_csv(), pandas.read_table(), pandas.read_excel(), and so on. The same methods exist for a CAS object as well. The only difference is that the methods are CAS table objects rather than a Pandas DataFrame.

Using the connection instance, you create the dm_inputdf DataFrame, which will contain the content of the training CAS table:

```
# Generate a data frame from input table
dm_inputdf = conn.CASTable(caslib = caslib, name = table).to_frame()
```

Python includes a built-in file type. Files can be opened by using the file type's constructor. You now open the _output.txt file for writing and print the first five rows of training data to the _output.txt file. This will be part of the Python output to display in the SAS Code node results.

```
# Create _output.txt file to print Python output
outF = open('_output.txt', 'w')

# Print first 5 rows from dm_traindf data frame
print(dm_traindf.head(), sep=' ', end='\n\n', file=outF, flush=False)
```

You create a DataFrame called X_train that contains the input variables and a series called y_train that contains the target variable, and you create a DataFrame called X_valid and a series called y_valid for the validation data. An evaluation set, eval_set, is also created to contain the training and validation data. In the following code, an XGB classifier model is created and fit to the training data set. The model is fit using the scikit-learn API and the model.fit() function. The eval_metric option is added to generate a plot of several metrics, such as error and log-loss, at each iteration of your evaluation set. You can see the parameters that the trained model uses by printing the model to the _output.txt file.

```
X_train = dm_traindf.loc[:,[%dm_input]]
y_train = dm_traindf['%qktrim(&dm_dec_vvntarget)']
X_valid = dm_validdf.loc[:,[%dm_input]]
y_valid = dm_validdf['%qktrim(&dm_dec_vvntarget)']
eval_set = [(X_train, y_train),(X_valid, y_valid)]
X = dm_inputdf.loc[:,[%dm_input]]

# Build Build XGBoost model
xgb = XGBClassifier()
xgb.fit(X_train, y_train, eval_metric=['error', 'logloss'],
eval_set=eval_set, verbose=True)
```

```
# Print GradientBoostingClassifier model in output file
print(xgb, sep=' ', end='\n\n', file=outF, flush=False)
outF.close()
```

You can make predictions by using the fit model on the complete data set. To make predictions, use the function predict_proba() as in the following code. You can add two new posterior variables created by using the predict_proba() to dm_inputdf DataFrame. The I_ variable, which is the categorical variable, is created using function predict() and added to the dm_inputdf DataFrame.

```
pred = xgb.predict_proba(X)
dm_inputdf['%qktrim(&dm_predicted_vvnvar)'] = pred[:,1]
dm_inputdf['P_BAD0'] = pred[:,0]
dm_inputdf['%qktrim(&dm_into_vvnvar)'] = pd.DataFrame(xgb.predict(X))

print(dm_inputdf.head(), sep=' ', end='\n', file=outF, flush=False)
outF.close()
```

As the following code shows, you use the plot_tree function that is provided by XGBoost to create and display the tree plot. It is important to change the size of the plot, because at the default size the plot details might not be legible. The num_trees option indicates which of the generated trees should be drawn, not the number of trees. Here the value is set to 3 so that you get the third tree that XGBoost generates. The current figure is saved to the rpt_tree.png file, and the resolution is specified in dots per inch while the figure is saved to the image file.

```
plot_tree(xgb,num_trees=3)

# Get current figure. This lets you get a reference to the current figure
when using pyplot.
fig = plt.gcf()

# Change the size of the plot.
fig.set_size_inches(15,10)

# Save the current figure. Specify the resolution in dots per inch while
saving to image file
fig.savefig('rpt_tree.png',dpi=700)
```

Then you upload the updated dm_inputdf data file to CAS running in SAS Viya and convert it to a CAS table, as follows. The upload_frame() method is used to upload the data file to the project caslib as the nodeid_score data set. This is the name of the scored table that Model Studio expects, and it uses this table to generate model assessment reports. You also want to upload the relative importance of score generated by the XGBoost table to CAS in the project library as a gb_varimp table by using the upload_frame() method.

```
# Upload a data file to CAS and parse it into a CAS table
conn.upload_frame(dm_inputdf, casout = dict(name = nodeid + '_score', caslib
= caslib, replace = True))

# Retrieve importance score for each feature and create a DataFrame
varimp = pd.DataFrame(list(zip(X_train, xgb.feature_importances_)),
columns=['Variable Name', 'Importance'])

# Upload a data file to CAS and parse it into a CAS table
```

```
conn.upload_frame(varimp, casout = dict(name = 'gb_varimp', caslib = caslib,
replace = True))
```

To take advantage of some of the reporting capabilities of Model Studio, you then convert the evaluation results from the evaluation set to a DataFrame called stat. This DataFrame is then uploaded and converted to create the gb_stat CAS table. This table contains four columns, which show the log-loss and error for both the training and validation data at each iteration.

```
 results = xgb.evals_result()
 stat = pd.DataFrame()
 stat['train_logloss'] = results['validation_0']['logloss']
 stat['valid_logloss'] = results['validation_1']['logloss']
 stat['train_error'] = results['validation_0']['error']
 stat['valid_error'] = results['validation_1']['error']
 conn.upload_frame(stat, casout = dict(name = 'gb_stat', caslib = caslib,
replace = True))
```

To transfer the _output.txt and rpt_tree.png files to the system where SAS Viya is running from the remote system where Python is running, Python Paramiko is used. Paramiko provides an abstraction of the SSHv2 protocol with both the client-side and server-side functionality. As a client, you can authenticate your identity by using a password or key, and as a server you can decide which users to allow access.

An instance of the Paramiko SSHClient can be used to make a connection to the remote server and transfer files. Paramiko requires you to validate that you trust the host machine you are connecting to. This validation is handled by calling the set_missing_host_key_policy() on the SSHClient and passing the policy that you want implemented when accessing a new remote machine. By default, the Paramiko SSHClient sets the policy to RejectPolicy, which rejects an attempted connection without validating. However, Paramiko does give you a way to implement a kind of "trust all" key policy—the AutoAddPolicy. Parsing an instance of the AutoAddPolicy to set_missing_host_key_policy() changes it to allow any host. If you try to make a connection without setting a key policy, you get the error "paramiko.ssh_exception.SSHException: Server 'hostname' not found in known_hosts."

The file transfers are handled by the Paramiko SFTPClient, which you get from calling oprn_sftp() on an instance of the Paramiko SSHClient. In the following code, use the put() method to upload files from a system running Python to the system running SAS Viya.

```
# Create an instance of SSHClient
ssh_client=paramiko.SSHClient()

# Set the policy to use when connecting to a server that doesn't have a host
key in either the system or locally. The missing host key
policy.AutoAddPolicy adds keys to this set and saves them when connecting to
a previously unknown server.
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Connect to an SSH server and authenticate to it. The server's host key is
checked against the system host key and any local host keys.
ssh_client.connect(hostname=&dm_cashost', username='&sysuserid')
ftp_client=ssh_client.open_sftp()
ftp_client.put('rpt_tree.png','&dm_nodedir&dm_dsep.rpt_tree.png')
ftp_client.put('_output.txt','&dm_nodedir&dm_dsep._output.txt')
```

Then you use a second CAS procedure step as follows to create a file called dm_srcfile.py that will run in the Python environment setup where the compute server is running SAS Viya. The remrunner (remote runner) package transfers a local script to a remote host and executes it. In this code, the gb01.py file that was created earlier is transferred to the remote system where Python is running, and the file is executed. In this case, the gb01.py file is copied to a temporary location (./remrunner/[PID]) on the remote host, its permissions are set to 0700, and the script is executed. During cleanup, the [PID] directory and all of its contents are removed before the connection is closed.

```
proc cas;
      file _codef "&dm_nodedir&dm_dsep.dm_srcfile.py";
      print "from remrunner import runner";
      print "r = runner.Runner(python.emd.sashq-d.openstack.sas.com',
'&sysuserid')";
      print "rval, stdout, stderr = r.run('&dm_nodedir&dm_dsep.gb01.py')";
      print "if rval:";
      print "    print(stderr)";
      print "else:";
      print "    print(stdout)";
      print "r.close()";
run;
```

If you want this scored table to be used by successor nodes, use the dmcas_register macro as follows to indicate that you want this table to remain with the project and to be used by any node that is connected to this SAS Code node:

```
/* Register _score table created in Python on remote host and uploaded to CAS
*/
%dmcas_register(dataset=&dm_output_memname, type=cas);
```

Finally, you create custom reports as follows that display tree plot and output files that are created on the remote Python system. You can also generate a report that displays the feature importance table that the XGBoost model generates. The dmcas_report macro is used to register and describe the tree plot image, Python output file, and feature importance table.

```
/* Display image file created in Python on remote host and transferred to
Viya system */
%dmcas_report(file=&dm_nodedir&dm_dsep.rpt_tree.png, reportType=Image,
description=%nrbquote('Tree Plot'), localize=N);

%dmcas_report(file=&dm_nodedir&dm_dsep._output.txt, reportType=CodeEditor,
type=TEXT, description=%nrbquote('Python Output'), localize=N);

/* Generate report to display feature importance table created by XGBoosts
model */

data &dm_lib..varimp;
   set &dm_datalib..gb_varimp;
run;

%dmcas_report(dataset=varimp, reportType=table,
description=%nrbquote(Gradient Boosting Classifier Feature Importance));
```

Note that for tables to be used in the reports, they must be located in the folder that is associated with the dm_lib libref. This is the node folder in the system where SAS Viya is installed. So, you download the gb_varimp CAS table to the system that is running Viya.

As you saw earlier, images and plots that are generated in Python and displayed in Model Studio can have a readability problem because of poor resolution. You now specify a report that can display either of the calculated metrics for the training and validation data.

You create the gbstat data set from the CAS gb_stat table by adding the iteration number and modifying the structure of the table so it can be plotted. The new variable, dataRole, identifies whether the statistic is a training or validation value. The logLoss and error variables contain the metric value corresponding to that iteration and the data role.

The dmcas_report macro gives users the ability to create more interactive reports than just images and tables. In the following code, you add the custom "XGB Iteration Plot" report to the results window of the SAS Code node. This report will be displayed as a line plot of either the logLoss or error metrics by iteration for both partition sets. The view option is used to indicate that you want to be able to select the displayed metric.

```
/* Create a the gbstat table to plot different metrics by iteration */
data &dm_lib..gbstat( keep=Iteration dataRole logLoss error);
   length Iteration 8 datarole $8 logLoss error 8;
   label dataRole='Data Role'; logLoss='Log Loss'; error = 'Error';
   set &dm_datalib..gb_stat;
   Iteration = _N_;
   dataRole='TRAIN';    logLoss = train_logloss;  error = train_error;
output;
   dataRole='VALIDATE'; logLoss = valid_logloss;  error = valid_error;
output;
run;

/* Request a series plot for each metric */
%dmcas_report(view=1, dataset=gbstat, comboDescription= XGB Iteration Plot,
reportType=SeriesPlot, description=%nrbquote(Log-Loss), X=Iteration,
y=logLoss, group=dataRole, localize=N);
%dmcas_report(view=2, dataset=gbstat, comboDescription= XGB Iteration Plot,
reportType=SeriesPlot, description=%nrbquote(Error), X=Iteration, y=error,
group=dataRole, localize=N);
```

When the node is run, the dm_srcfile.py file is executed in the Python environment on the system where SAS Viya is running. The remrunner code is executed on the compute server on the Viya system, and the gb01.py file is submitted to the remote Python server. The Python code that is executed on the remote system and its results are transferred using Paramiko package methods. The image file with the tree plot and the output.txt files are copied to the node directory from the remote system, and the variable importance CSV file is uploaded to CAS as a CAS table. The _score table is uploaded to CAS using SWAT, and the assessment results are calculated from the _score table.

The Node tab shown in Figure 10 contains the tree plot image file, the variable importance from XGBoost, Python code, and Python output.

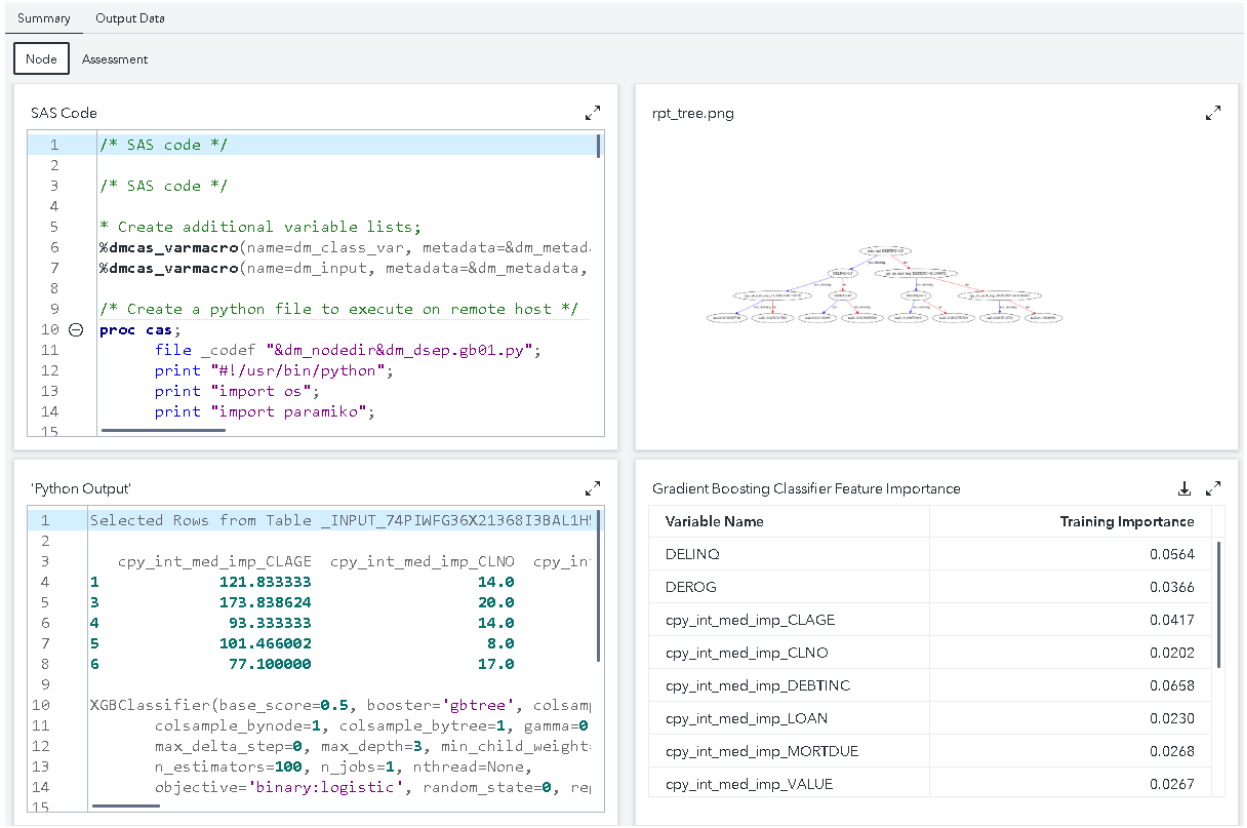

Figure 10. SAS Code Node Results

The XGB Iteration Plot report shown in Figure 11 also appears on the Node Results tab. The selector enables you to control whether to display the log-loss or the error.

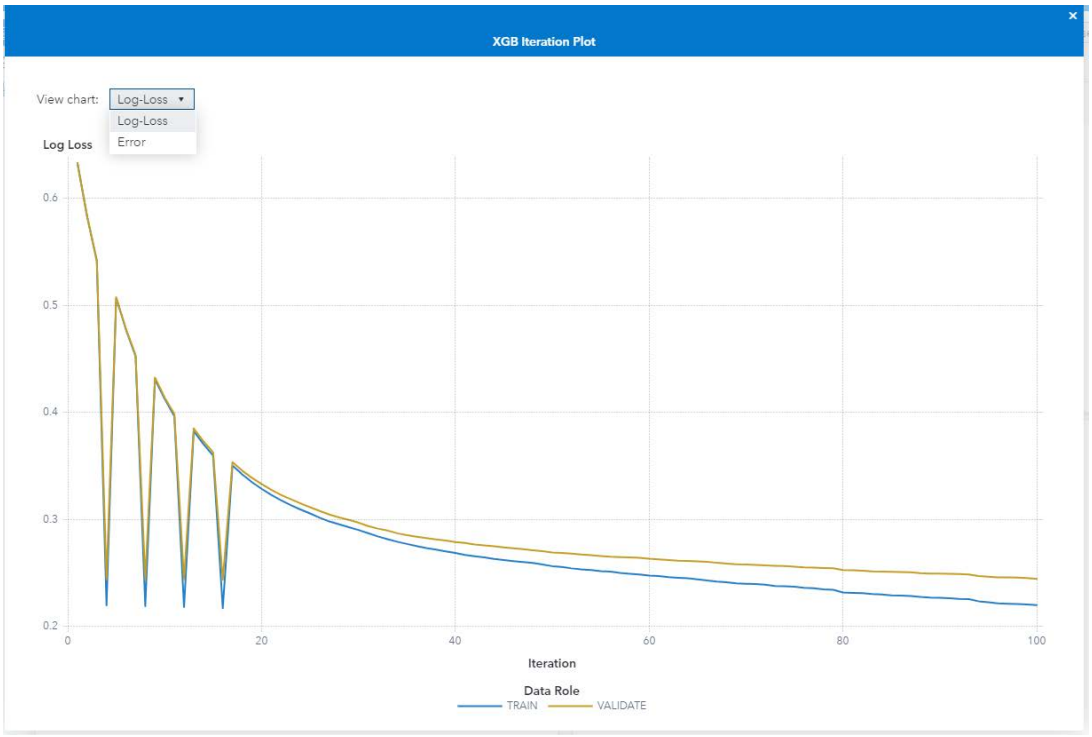

Figure 11. Custom Fit Statistics Report

The Assessment tab in Figure 12 displays the assessment reports: lift reports, ROC reports, fit statistics table, and event classification chart. The fit statistics table shows that the misclassification rate is 0.0822 for the training partition, 0.0979 for the validation partition , and 0.0990 for the test partition. Note that because Python generated a scored table, the only model interpretability report available is the surrogate variable importance report.

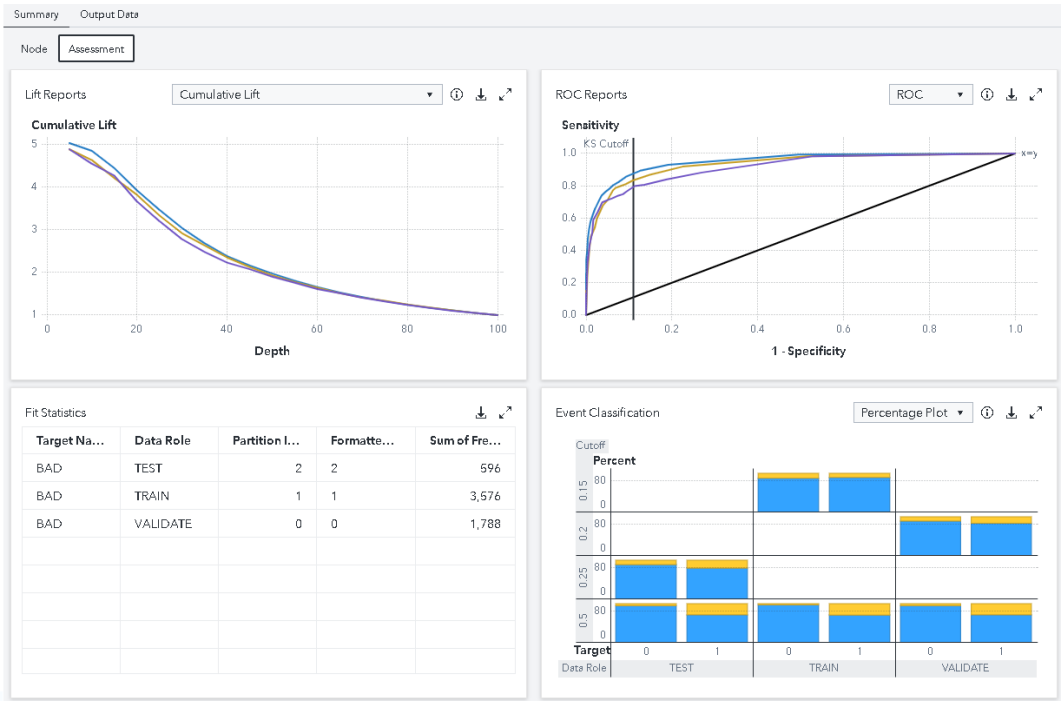

Figure 12. SAS Code Assessment Results

As mentioned earlier, the current limitation of remrunner is that it assumes that SSH keys to allow password-free log-ins are already in place. There is no option to provide a password.

If you don't have password-free log-ins in place, you can use an instance of the Paramiko SSHClient to connect to the remote server and transfer files. Paramiko requires you to validate that you trust the host machine you are connecting to. This validation is handled by calling the set_missing_host_key_policy() on the SSHClient and passing the policy that you want implemented when you access a new remote machine. By default, the Paramiko SSHClient sets the policy to RejectPolicy, which rejects the connection without validating. However, Paramiko does give you a way to use a kind of "trust all" key policy—the AutoAddPolicy. Parsing an instance of the AutoAddPolicy to set_missing_host_key_policy() changes it to allow any host.

The file transfers are handled by the Paramiko SFTPClient, which you get from calling oprn_sftp() from an instance of the Paramiko SSHClient. You use the put() method to upload files from the Python system to the system where SAS Viya is running. To run the command, the exec_command is called on the SSHClient instance; for this example, the command is `'python /tmp/jk/gb01.py'`. The response is returned as a tuple (stdin, stdout, stderr).

Instead of using remrunner, you could use the SSHClient to connect to the remote machine. In the following code, an FTP connection is opened to transfer the file to the remote system, and the exec_command is used to submit the Python file on the remote system:

```
proc cas;
      file _codef "&dm_nodedir&dm_dsep.dm_srcfile.py";
      print "import paramiko";
```

```
      print "import sys";
      print "ssh_client=paramiko.SSHClient()";
      print
          "ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())";
      print "ssh_client.connect(
            hostname='python.emd.sashq-d.openstack.sas.com', username='root',
            password='rootpwd', look_for_keys=False)";
      print "ftp_client=ssh_client.open_sftp()";
      print
ftp_client.put('&dm_nodedir&dm_dsep.gb01.py','/tmp/usr/gb01.py')";
      print "ftp_client.close()";
      print "stdout = ssh_client.exec_command('python /tmp/usr/gb01.py')[1]";
      print "for line in stdout:";
      print "    print(line)";
      print "ssh_client.close()";
run;
```

CONCLUSION

Model Studio gives you the ability to execute Python scripts to incorporate custom functionality in a Model Studio pipeline. The Open Source Code node lets you specify Python code directly and provides data items to facilitate the integration of the node in a pipeline. The data, which in most cases are a sample, are downloaded to the system where SAS Viya is running and then converted to a Pandas DataFrame that can be used by Python packages. The Open Source Code node framework enables you to display plots and tables that are produced in Python. If the Open Source Code node is a Supervised Learning node, then creating the appropriate scored data frame enables Model Studio to produce assessment reports. Code that you provide in the Open Source Code node runs in a Python environment that is on the compute server where SAS Viya is running. You can also use the SAS Code node to run Python scripts. Model Studio provides a set of macros and macro variables that can be used for integration. You can use the SWAT package to execute CAS actions on the CAS server and to produce reports to be displayed in the node results. You can also use SWAT to call packages like DLPy to take advantage of CAS actions and Python functionality. Finally, you can run a Python script on a remote server and use SWAT to connect to an existing CAS session to access the data, run a Python modeling package, and upload resulting tables into CAS. Using the SSHClient, you can transfer files from the remote server to the system where SAS Viya is running in order to display reports in the SAS Code node results window.

REFERENCES

DLPy. SAS Deep Learning Python Interface Document. Available at https://sassoftware.github.io/python-dlpy/.

Luebbert, J., and Myneni, R. (2019). "SAS and Open Source: Two Integrated Worlds." In *Proceedings of the SAS Global Forum 2019 Conference.* Cary, NC: SAS Institute Inc. Available at https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3415-2019.pdf.

SAS Developer Home. Resources for Developers on SAS and Open Source. https://developer.sas.com/home.html.

Smith, K., and Meng, X. (2017). "Using Python with SAS Cloud Analytic Services (CAS)." In *Proceedings of the SAS Global Forum 2017 Conference.* Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings17/SAS0152-2017.pdf.

Smith, K., and Meng, X. (2017). *SAS Viya: The Python Perspective*. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The authors would like to thank Radhikha Myneni and Ed Huddleston for their helpful advice and assistance with this paper.

CONTACT INFORMATION

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

Jagruti Kanjia
SAS Institute Inc.
Jagruti.Kanjia@sas.com

Dominique Latour
SAS Institute Inc.
Dominique.Latour@sas.com

Jesse Luebbert
jesse.luebbert@gmail.com

APPENDIX

DATA ITEMS FOR THE OPEN SOURCE CODE NODE

The data items in the following table are available for certain expected cases in your Python code.

Data Item	Description
dm_class_input	A list of names that identify the categorical (nominal and ordinal) variables. The data type is a list in Python.
dm_classtarget_intovar	A variable that identifies the categorical variable for a categorical target. The data type is a string.
dm_classtarget_level	A list of values that identify the various levels in the target variable. The data type is a list in Python.
dm_dec_target	A variable that identifies the name of the target. The data type is string.
dm_input	A list of names that identify the input (interval and categorical) variables. The data type is a list in Python.
dm_inputdf	A data frame that contains the sampled input data. This is available only when the Generate data frame property is selected. The data type is a Pandas DataFrame in Python.
dm_interval_input	A list of names that identify the interval variables. The data type is a list in Python.
dm_model	An object that identifies the model. The usage of this object is optional but recommended for future use.
dm_nodedir	A variable that identifies the node's working directory—that is, where the local files are created. The data type is string.
dm_partition_train_val	A variable that identifies the value that identifies the training partition. The data type is string or integer.
dm_partitionvar	A variable that identifies the name of the data partition. The data type is string.
dm_predictionvar	A list of names that identify the prediction variables. The data type is a list in Python.
dm_scoreddf	A data frame that you must create with model predictions when you execute the node in the Supervised Learning group. This item must be created when the Generate data frame property is selected and must contain the same number of rows as dm_inputdf. The data type is a Pandas DataFrame in Python.
dm_traindf	A data frame that contains the sampled training data. This is available only when the Generate data frame property is selected. The data type is a Pandas DataFrame in Python.
node_data.csv	A file in comma-separated value format that identifies the sampled input data in the dm_nodedir directory.

Data Item	Description
node_scored.csv	A file in comma-separated value format that you must create by using model predictions when you execute the node in the Supervised Learning group and model assessment is performed. This file must be created when Generate data frame is not selected and must contain the same number of rows as node_data.csv.

MACRO VARIABLES AND MACROS FOR THE SAS CODE NODE

The macros in the following table are available for use in your SAS training code.

Macro Group	Macro Name	Macro Description
Variables	dm_binary_input	Identifies the n-literal names of the binary input variables
	dm_dec_target	Identifies the n-literal name of the target variable
	dm_id	Identifies the n-literal names of the ID variables
	dm_interval_input	Identifies the n-literal names of the interval input variables
	dm_into_var	Identifies the n-literal name of the categorical variable for a categorical target variable. This macro is defined, but it is empty for an interval target variable.
	dm_key	Identifies the n-literal name of the key variable
	dm_predicted_var	Identifies the n-literal names of the posterior probability variables for a categorical target variable or the predicted variable for an interval target variable
	dm_nominal_input	Identifies the n-literal names of the nominal input variables
	dm_offset	Identifies the n-literal name of the offset variable
	dm_ordinal_input	Identifies the n-literal names of the ordinal input variables
	dm_segment	Identifies the n-literal names of the segment variables
	dm_text	Identifies the n-literal names of the text variables
	dm_unary_input	Identifies the n-literal names of the unary input variables with missing values
Utility	dmcas_checkMacroVar	Checks the existence of a macro variable. A default value is assigned if the macro variable does not exist.
	dmcas_copybinaryfile	Copies a binary file

Macro Group	Macro Name	Macro Description
	dmcas_copyfile	Copies a text file
	dmcas_metachange	Specifies changes to the metadata
	dmcas_register	Registers files, data sets, and CAS tables
	dmcas_report	Registers and defines reports

The following macro variables are defined and available for use in your SAS training code.

Macro Variable Group	Macro Variable	Macro Variable Description
CAS: Files	dm_data	Identifies the CAS training table
	dm_datalib	Identifies the libref associated with the training table
	dm_data_caslib	Identifies the caslib associated with the training table
	dm_data_outmodel	Identifies the remote CAS outmodel table
	dm_data_rstore	Identifies the remote analytic store
	dm_ds_caslib	Identifies the CAS library of the project source table
	dm_memname	Identifies the name of the training table (that is, a one-level name)
	dm_memnamenlit	Identifies the n-literal name of the training table (that is, a one-level name)
	dm_outmodeltable	Identifies the name of the remote CAS outmodel table (a one-level name)
	dm_output_data	Identifies the CAS output table. This table is used to produce assessment reports for the Supervised Learning node.
	dm_output_memname	Identifies the name of the output table (that is, a one-level name)
	dm_projectData	Identifies the project CAS source table
	dm_rstoretable	Identifies the name of the remote analytic store associated with the node (a one-level name)
CAS: General	dm_cashost	Identifies the host name of the server that is associated with the CAS session
	dm_caslib	Identifies the caslib associated with the training table
	dm_casport	Identifies the port number of the server associated with the CAS session

Macro Variable Group	Macro Variable	Macro Variable Description
	dm_cassessionid	Identifies the session ID of the CAS session
	dm_cassessref	Identifies the name of the CAS session that you want to connect to
Data: Partition	dm_partitionvar	Identifies the name of the partition variable. This macro variable is blank if the data are not partitioned. This macro variable is _partInd_ if the partitioning is done by Model Studio. If a partition variable is specified on the Data tab, then this macro variable is the name of the specified partition variable.
	dm_partitiontrainwhereclausenlit	Identifies the WHERE clause based on the n-literal name of the partition variable that can be applied to retrieve the training observations (for example, `_PartInd_'n = 1)
	dm_partitiontestwhereclausenlit	Identifies the WHERE clause based on the n-literal name of the partition variable that can be applied to retrieve the test observations (for example, `_PartInd_'n = 2)
	dm_partitionvalidwhereclausenlit	Identifies the WHERE clause based on the n-literal name of the partition variable that can be applied to retrieve the validation observations (for example, `_PartInd_'n = 0)
	dm_partition_statement	Identifies the partition statement
	dm_partition_test_val	Identifies the value of the partition variable that corresponds to the test observations
	dm_partition_train_val	Identifies the value of the partition variable that corresponds to the training observations
	dm_partition_valid_val	Identifies the value of the partition variable that corresponds to the validation observations

Macro Variable Group	Macro Variable	Macro Variable Description
Data: Target	dm_data_targetInfo	Identifies the SAS data set that contains information about the target variable. For a categorical target variable, this data set contains the level values and frequencies, the associated posterior probability values, and the name of the categorical variable. For an interval target variable, the data set contains the name of the predicted variable.
	dm_dec_event	Identifies the event level of a categorical target variable. By default, the event is the first level as defined by the order of the target variable. You can also select the event level of a categorical target variable on the Data tab.
	dm_dec_format	Identifies the format of the target variable
	dm_dec_label	Identifies the label of the target variable
	dm_dec_level	Identifies the measurement level of the target variable. Possible measurement levels are BINARY, INTERVAL, ORDINAL, and NOMINAL.
	dm_dec_nonevent	Identifies the nonevent value for a binary target
	dm_dec_order	Identifies the order of the target variable. Possible values are ASCFMT, ASC, DESC, and DESFMT. The default value is DESC.
	dm_dec_type	Identifies the type (C or N) of the target variable
	dm_dec_vvntarget	Identifies the name of the target variable
	dm_into_vvnvar	Identifies the categorical variable for a categorical target
	dm_predicted_vvnvar	Identifies the prediction variable for an interval target or the posterior variable associated with the target event for a categorical target
Data: Variables	dm_key	Identifies the name of the key variable
	dm_num_binary_input	Identifies the number of binary input variables
	dm_num_id	Identifies the number of ID variables

Macro Variable Group	Macro Variable	Macro Variable Description
	dm_num_interval_input	Identifies the number of interval input variables
	dm_num_nominal_input	Identifies the number of nominal input variables
	dm_num_ordinal_input	Identifies the number of ordinal input variables
	dm_num_segment	Identifies the number of segment variables
	dm_num_text	Identifies the number of text variables
	dm_num_unary_input	Identifies the number of unary input variables that have missing values. Variables that are constant without missing values are assigned the role Rejected.
	dm_projectmetadata	Identifies the SAS data set that contains the project metadata
Node: General	dm_codebar	Identifies a code separator
	dm_dsep	Identifies the operating system file delimiter (for example, "\" for Windows)
	dm_labelid	Identifies a number that is used to construct array names and statement labels in the generated DS1 code. This is used to prevent naming issues when you combine models.
	dm_maxnamelen	Identifies the maximum length for the name of a new variable
	dm_nodeguid	Identifies the global identifier of the node. This identifier is used to register tables and results.
	dm_nodeid	Identifies the node ID
Node: Local Files	dm_data_outfit	Identifies the fit statistics data set
	dm_data_scorevar	Identifies the data set to be used to register individual variable transformations
	dm_file_astore	Identifies the local analytic store
	dm_file_deltacode	Identifies the file containing the DATA step code that modifies the columnsmeta metadata that is exported by the node
	dm_file_scorecode	Identifies the file that contains the score code

Macro Variable Group	Macro Variable	Macro Variable Description
	dm_file_traincode	Identifies the file that contains the training code
	dm_folder_scorevar	Identifies the local folder that contains the score code files. The score code files contain individual variable transformations.
	dm_lib	Identifies the SAS library where components should save and create local tables
	dm_metadata	Identifies the local data set that contains the imported metadata for the node
	dm_nodedir	Identifies the folder where local files are created