

Paper SAS4536-2020

Choose Your Own Adventure:

Manage Model Development via a Python IDE

Jon Walker, SAS Institute Inc.

ABSTRACT

Data scientists often need to work with multiple languages and in multiple analytic environments to solve a problem. SAS® provides a complete end-to-end environment, but it has traditionally been accessible to users only through GUIs and SAS languages. This paper introduces a new tool enabling data scientists to manage components of the analytics life cycle from within any Python environment. We first demonstrate how to register a model developed with Python using SAS® Model Manager, before exploring methods for managing, deploying, and tracking the model. In addition, we show how to accomplish supporting tasks such as rendering visualizations and extending the existing functionality.

INTRODUCTION

In the past few years, the Python language has quickly become the preferred language for many data scientists (Mitchell 2019).

Although SAS and Python are sometimes viewed as competing technologies, the reality is that these technologies can complement each other quite well. The SAS platform contains numerous tools to help users manage the entire analytics lifecycle and a collection of high-performance algorithms designed to scale to large data, while Python has a huge user community and a wide selection of packages that make it an ideal language for integrating different technologies. It's only natural that Python users would want to leverage some of the additional functionality in SAS.

This paper introduces the new `sasctl` package for Python, designed to allow control of the SAS® Viya® platform from a Python runtime. It can be used as a Python module or executed directly from a command line interface. There are already several excellent Python packages available for building analytic models (Pedregosa, et al. 2011, Smith and Meng 2017), but this is not one of them. Instead, the `sasctl` package is designed to complement these analytics packages. This paper focuses on activities often related to but separate from model building.

Specifically, we first demonstrate how to use `sasctl` for managing model registration and deployment of both SAS and open-source models. Then, we introduce additional functionality such as monitoring model performance and rendering visualizations.

This paper should be relevant to data scientists, developers, analysts, or anyone else who needs to communicate with the SAS Viya platform and prefers Python. The examples covered are intended to be simple, but a basic understanding of the Python language is assumed. Additionally, knowledge of standard analytics packages like SAS SWAT and `scikit-learn` is not required but might be helpful.

All code examples in this paper use `sasctl` v1.5 and are available on the `sasctl` GitHub page. (SAS Institute Inc. 2020a.)

MODEL MANAGEMENT

An easy way to get a gentle introduction to using sasctl is to perform a few of the most common tasks for data scientists – model registration, deployment, and execution. Conveniently, these are also the areas where sasctl currently affords the highest levels of abstraction and ease of use.

SWAT MODEL

The following example demonstrates how to use sasctl to easily manage a model built with SAS. We use the SWAT package to define a simple regression model on the well-known Boston housing data set (Belsley, Kuh, and Welsch 1980). After training the model, we demonstrate how to easily register it with SAS Model Manager. This allows the model, and any associated metadata, to be stored in a central repository, version-controlled, and tracked over time.

```
1 import swat
2 from sasctl import Session
3 from sasctl.tasks import register_model, publish_model
4
5 s = swat.CAS('example.sas.com', 5570, 'arthur', 'K1ng0fTheBr!tons')
6 s.loadactionset('regression')
7 tbl = s.upload('data/boston_house_prices.csv').casTable
8
9 features = list(tbl.columns[tbl.columns != 'medv'])
10 tbl.glm(target='medv', inputs=features, savestate='model_table')
11 astore = s.CASTable('model_table')
```

In the code above, lines 5-7 establish a connection to SAS® Cloud Analytic Services (CAS), load the regression package, and import the data set from a local CSV file. Lines 9-11 fit the regression model to the data and save the results. We won't examine the SWAT package in detail here, but for more information see (Smith and Meng 2017, SAS Institute Inc. 2020b). The key point is that the end result is a small, binary artifact, or *ASTORE*, containing the final model, and this is what we provide to sasctl to register:

```
12 with Session('example.sas.com', 'arthur', 'K1ng0fTheBr!tons'):
13     model = register_model(astore, 'Linear Regression', 'Boston Housing', force=True)
```

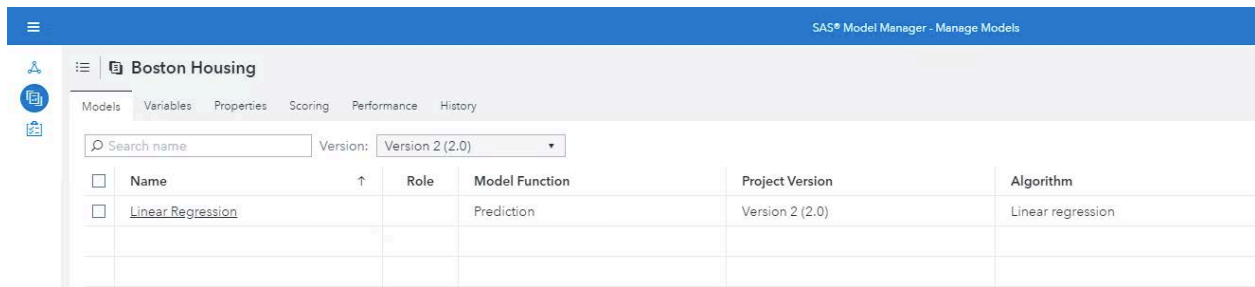
Everything in sasctl requires an established session to a SAS Viya server. At its most basic level, sasctl is a sophisticated REST client that calls the REST APIs available in all SAS Viya environments. Creating a session allows sasctl to repeatedly call those APIs on your behalf without requiring you to authenticate each time. In this case, we're establishing a session on line 12 using the same credentials we used to connect to CAS. There are a variety of ways to establish a session. See the APPENDIX for more details and for solutions to common problems (like SSL errors).

Once a session has been created, it is used by default for all subsequent tasks without explicitly referencing it. The next step is to call the *register_model* task (line 13) and provide:

1. the actual model to put in SAS Model Manager
2. a name for the model
3. the name of the project in which to create the model

In this case, we're using the ASTORE model, creating a project called *Boston Housing*, and naming our model *Linear Regression*. The optional `force=True` parameter instructs `sasctl` to automatically create the *Boston Housing* project if it does not already exist.

At this point our model should now be registered in SAS Model Manager and should be similar to Display 1.



The screenshot shows the SAS Model Manager interface for a project named 'Boston Housing'. The interface includes a search bar, a version dropdown set to 'Version 2 (2.0)', and a table of models. The table has columns for Name, Role, Model Function, Project Version, and Algorithm. One model is listed: 'Linear Regression' with a role of 'Prediction', project version 'Version 2 (2.0)', and algorithm 'Linear regression'.

Name	Role	Model Function	Project Version	Algorithm
Linear Regression		Prediction	Version 2 (2.0)	Linear regression

Display 1. Model in SAS Model Manager

Now that the model is registered, we can use it with the full range of SAS Model Manager capabilities. We won't go into those details here, but for more information about the available features, see the SAS Model Manager documentation (SAS Institute Inc. 2019g, SAS Institute Inc. 2019a).

Of course, the next logical step is to publish the model somewhere and then run it. For this, we'll push our model to the SAS® Micro Analytic Service (SAS Institute Inc. 2019f), a light-weight engine designed for real-time scoring of records:

```
14     module = publish_model(model, 'maslocal')
15
16     first_row = tbl.head(1)
17     module.score(first_row)
```

As you can see from line 14, we publish the model with just a single line of code. We provide the model and the name of a publishing destination. Here, we choose `maslocal`, the default SAS Micro Analytic Service instance available in most SAS Viya environments. The result is a newly created SAS Micro Analytic Service module, decorated with Python methods corresponding to operations available with our model.

Since SAS Micro Analytic Service is a real-time scoring service, it expects a single row of data at a time, so line 16 selects the first row of data from our data set and then "scores" the record by calling SAS Micro Analytic Service. The result is shown in Output 1 and is the prediction from our model on that input:

```
30.003843377
```

Output 1. Predicted Median Value (in \$1,000s)

And with that, we've trained a new model, registered it in our repository, deployed it in a real-time environment, and successfully executed it, all with just a few lines Python code.

SCIKIT-LEARN MODEL

This next example is similar to the previous one. However, this time we work with a model built using the open-source `scikit-learn` package (Pedregosa, et al. 2011) rather than

building it with SAS algorithms. Just as before, the first step in the process is to train the model:

```
1 import swat
2 import pandas as pd
3 from sasctl import Session, register_model, publish_model
4 from sasctl.services import model_publish as mp
5 from sklearn.ensemble import GradientBoostingRegressor
6
7 df = pd.read_csv('data/boston_house_prices.csv')
8
9 target = 'medv'
10 X = df.drop(target, axis=1)
11 y = df[target]
12
13 model = GradientBoostingRegressor()
14 model.fit(X, y)
```

In lines 7-11 we import the Boston housing data set using Pandas (Reback, et al. 2019) before separating the data into X and y variables containing an array of input features and the target output, respectively. Line 13 defines a gradient boosting model, and line 14 trains the model on our housing data set. At this point we have a simple, but complete model ready for registration and deployment. Despite not being a SAS model, we register this second model in SAS Model Manager using the same `register_model` task we used before:

```
16 with Session('example.sas.com', username='BlackKnight', password='invincible!'):
17     model_name = 'Gradboost Regression'
18
19     register_model(model, model_name, input=X, project='Boston Housing', force=True)
```

Line 16 creates a connection to the SAS Viya environment. Line 19 registers the model into SAS Model Manager and stores it in the same project as the previous model. Note that because the project has already been created, the `force=True` option has no effect. Unlike SAS models, those produced with scikit-learn do not contain information about the model inputs and outputs. Besides being good to document, this information is critical if we want to execute the model or track model degradation over time. The `input=` parameter provides this information and the easiest way to do it is to provide the training data set. Behind the scenes sasctl analyzes the data set to determine variable names and types as well as run a sample of the data through the model to determine output variables.

Display 2 shows the updated SAS Model Manager project with the new Python model alongside the first model.

Name	Role	Model Function	Project Version	Algorithm
Gradboost Regression		Prediction	Version 2 (2.0)	GradientBoostingRegressor
Linear Regression		Prediction	Version 2 (2.0)	Linear regression

Display 2. Updated SAS Model Manager Project

If you open the new model and explore, you'll notice a few things. First, sasctl has automatically created and uploaded the following collection of files to accompany the model that should be similar to those in Display 3:

- a pickled copy of the model.
- a *requirements.txt* file that lists the Python packages installed in the environment where we registered the model.

Note: The goal is to capture exactly which versions of packages might have been used in building the model. Unfortunately, this is just an estimate, as we can only see which packages are installed, not necessarily which ones were used. This is a good baseline, but you might refine this list as necessary for production models.

- SAS programs that wrap the Python model in SAS DS2 code, which enables more of the SAS components to interact with a model that was not built with SAS (SAS Institute Inc. 2019d).

```

1 package _EBD88AB7F23E4D3B95684419FDDC1B0 / overwrite=yes;
2   dcl package pyamas py;
3   dcl package logger logr('App.tk.MAS');
4   dcl varchar(67108864) character set utf8 pycode;
5   dcl int revision;
6
7   method init();
8
9     dcl integer rc;
10    if null(py) then do;
11      py = _new_ pyamas();
12      rc = py.useModule('EBD88AB7F23E4D3B95684419FDDC1B02', 1);
13      if rc then do;
14        rc = py.appendSrcLine('try:');
15        rc = py.appendSrcLine('  import pickle, base64');
16        rc = py.appendSrcLine('  bytes = b"gANjc2tsZWYybi51bnNlbWJs');
17        rc = py.appendSrcLine('  obj = pickle.loads(base64.b64deco

```

Display 3. Files Uploaded to SAS Model Manager

Display 4 shows the input and output parameters that sasctl determined and presented to SAS Model Manager because the *input=* parameter was provided when registering the model. In addition to being good practice, this is also necessary if we wish to track the model's performance over time.

SAS® Model Manager - Manage Models

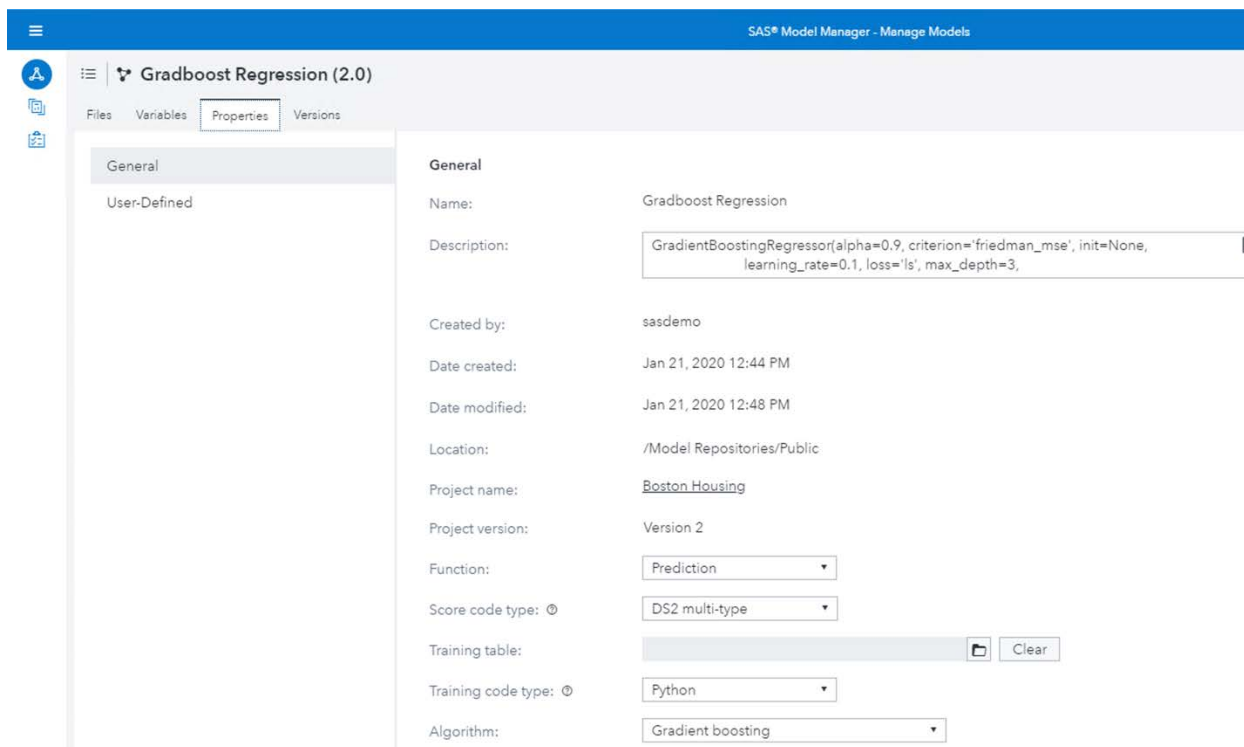
Gradboost Regression (2.0)

Files Variables Properties Versions

<input type="checkbox"/>	Name	Data Type	Input	Output
<input type="checkbox"/>	age	⊕ Decimal	✓	
<input type="checkbox"/>	b	⊕ Decimal	✓	
<input type="checkbox"/>	chas	⊕ Integer	✓	
<input type="checkbox"/>	crim	⊕ Decimal	✓	
<input type="checkbox"/>	dis	⊕ Decimal	✓	
<input type="checkbox"/>	indus	⊕ Decimal	✓	
<input type="checkbox"/>	lstat	⊕ Decimal	✓	
<input type="checkbox"/>	nox	⊕ Decimal	✓	
<input type="checkbox"/>	ptratio	⊕ Decimal	✓	
<input type="checkbox"/>	rad	⊕ Integer	✓	
<input type="checkbox"/>	rm	⊕ Decimal	✓	
<input type="checkbox"/>	tax	⊕ Integer	✓	
<input type="checkbox"/>	zn	⊕ Decimal	✓	
<input type="checkbox"/>	var1	⊕ Decimal		✓

Display 4. Input and Output Variables in SAS Model Manager

And finally, sasctl extracts and stores some additional metadata about the model, including the type of algorithm used and a description. SAS Model Manager allows user-defined properties that are searchable, so sasctl also includes the model parameters and Python package information in these properties. Users can find models with specific settings, or models that were built using a particular package version.



Display 5. Model Properties in SAS Model Manager

Of course, the files and metadata included with the model allow for customization. By default, `sasctl` includes this information for the sake of completeness, and to ensure SAS Model Manager has sufficient information to allow it to interact with the model that was not built with SAS.

Because of this, we have the ability to publish the model just as if it were a SAS model. The previous example demonstrated how to publish a model to SAS Micro Analytic Service, the real-time scoring engine. In the following example, we'll demonstrate publishing a model to CAS, the distributed analytics engine, that affords large-scale data processing capabilities to the SAS platform.

```

20     if mp.get_destination('caslocal') is None:
21         mp.create_cas_destination('caslocal', 'Public', 'model_table')
22
23     module = publish_model(model_name, 'caslocal')
```

Some environments might already have a CAS publishing destination, while others might not. Lines 20 and 21 define such a destination, called `caslocal` if it does not already exist. The specific parameters on line 21 dictate that models published to this destination will be stored in a table named `model_table`, located in the `Public` caslib (SAS Institute Inc n.d.a). Line 23 publishes the model to CAS and makes it available for execution.

Behind the scenes, what's actually being published is a DS2 program that wraps our Python model. This is because CAS doesn't currently know how to execute Python code directly, but it uses the PyMAS package (SAS Institute Inc 2018b) in DS2 to handle the execution. Note that the example above assumes that the environment has been configured for PyMAS execution (SAS Institute Inc. 2018a), which is beyond the scope of this paper.

Once published, the model executes like any other CAS model. For this, we will use SWAT to connect to CAS and run the model:

```

24 cas = swat.CAS('example.sas.com', 5570, 'BlackKnight', 'invincible!')
25 tbl = cas.upload(X).casTable
26 cas.loadactionset('modelpublishing')
27
28 result = cas.runModelLocal(modelName=module.name,
29                             modelTable=dict(name='model_table', caslib='Public'),
30                             inTable=tbl,
31                             outTable=dict(name='boston_scored'))
32
33 cas.CASTable('boston_scored').head()

```

Lines 24-26 are very similar to the initial steps in the SWAT example covered previously – they establish a connection to CAS and load the necessary data and CAS action sets.

Lines 28-31 contain a single command but are spread out for readability. We execute the `runModelLocal` CAS action (SAS Institute Inc 2019b) to score the model on the uploaded input data and write the results to a CAS table named `boston_scored`.

Line 33 retrieves the first five rows of scored output from the CAS table, which should appear similar to those shown in Output 2.

var1	crim	zn	indus	chas	nox	...	dis	dis	tax	ptratio	b	lstat
25.916	0.006	18	2.31	0	0.538	...	4.090	1	296	15.3	396.9	4.98
21.963	0.027	0	7.07	0	0.469	...	4.967	2	242	17.8	396.9	9.14
33.927	0.027	0	7.07	0	0.469	...	4.967	2	242	17.8	392.8	4.03
34.145	0.032	0	2.18	0	0.458	...	6.0622	3	222	18.7	394.6	2.94
35.413	0.069	0	2.18	0	0.458	...	6.0622	3	222	18.7	396.9	5.33

Output 2. Sample Results from a CAS Table with “var1” Holding a Predicted Median Value (in \$1,000s)

SUPPORTING TASKS

In the previous section we demonstrated how `sasctl` enables Python developers to easily integrate with SAS and accomplish some of the most common tasks in data science. In this section, we’ll demonstrate how to achieve some less common, but equally useful tasks.

PERFORMANCE MONITORING

While registering and deploying models are obviously critical steps in any analytics pipeline, there are also a host of challenges that only surface once a model is in production. One of these, model degradation, is crucial to manage. Over time almost all models will degrade, whether it’s because the process being modeled changes (for example, shifting user behavior) or because the input data changes (for example, shifting demographic data). If we can monitor these changes over time, then we can intelligently determine when to retrain our model. SAS Model Manager performs this monitoring and provides helpful visualizations over time (SAS Institute Inc. 2019a). The following example demonstrates using this functionality on a scikit-learn model. The following example code builds on top of the scikit-learn model developed in the previous section.

```

33 from sasctl import update_model_performance
34 from sasctl.services import model_management as mm, model_repository as mr
35
36 project = mr.get_project('Boston Housing')

```



```

37     project['targetVariable'] = target
38     project = mr.update_project(project)
39
40     mm.create_performance_definition(model_name, 'Public', 'boston')
41
42     perf_df = X.copy()
43     perf_df['var1'] = model.predict(X)
44     perf_df[target] = y
45
46     for period in ('q1', 'q2', 'q3', 'q4'):
47         sample = perf_df.sample(frac=0.2)
48         update_model_performance(sample, model_name, period)

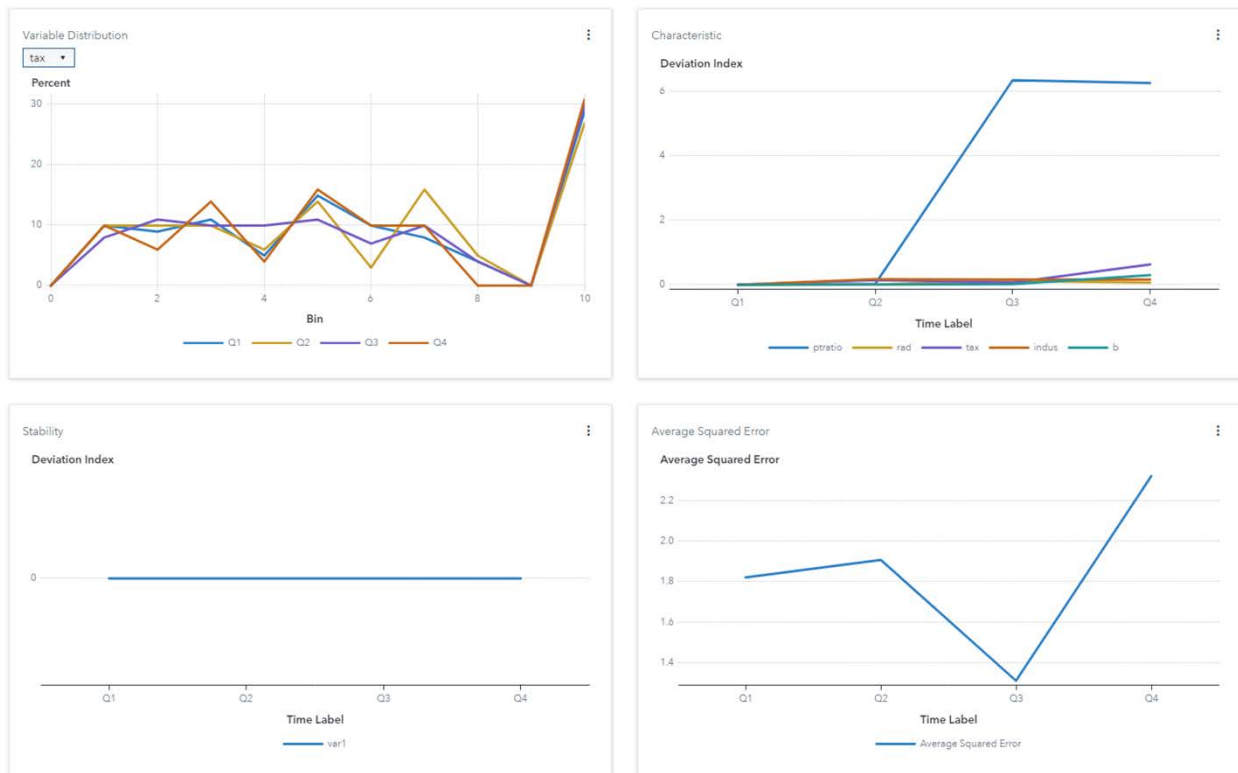
```

Up until now, we've only dealt with high-level sasctl tasks, not the underlying services supporting those tasks. Here we'll use two services directly: the *model_management* and *model_repository* services. Lines 33 and 34 import those services as well as the *update_model_performance* task, which we will use shortly.

SAS Model Manager monitors model performance by inspecting data tables containing the model inputs and outputs. However, before it can do that it must know which column in the table contains the target value. In lines 36-38 we use the *model_repository* service to update the model project and specify the column containing the target variable.

Line 40 uses the *model_management* service to create a performance definition. Here we specify the name of the model to monitor and tell SAS Model Manager we'll be placing the relevant data tables in the *Public* caslib with a *boston* prefix. Typically, we would collect the model's output over time once it's deployed and feed this data to SAS Model Manager, but for demonstration purposes we're going to mockup this data. Lines 42-44 create a new data set containing the model inputs, the actual target value, and the model's output for each input.

In lines 46-48 we repeatedly take a 20% sample from this data set and use it to represent the results of the model for each quarter. As each result is uploaded to SAS Model Manager the model metrics and visualizations are automatically recomputed, resulting in a set of visualizations similar to those shown in Display 6.



Display 6. Model Manager Performance Reports

REPORT VISUALIZATIONS

It is also possible to retrieve reports and visualizations from SAS, rendered on the fly for the desired size. SAS includes two microservices that manage reports (SAS Institute Inc. n.d.e) and the display of their contents (SAS Institute Inc. n.d.d) and sasctl leverages these to allow easy rendering of report visualizations.

```

1 from sasctl import Session
2 from sasctl.services import reports, report_images
3
4 Session('example.sas.com', 'knight', 'Ni!')
5
6 activity_report = reports.get_report('CAS Activity')
7
8 elements = reports.get_visual_elements(activity_report)
9 graph = next(e for e in elements if e.label == 'I/O and Threads')
10
11 report_images.get_images(activity_report, elements=graph)

```

Currently, no high-level task exists in sasctl for retrieving report content. Despite this, it is still a straightforward process to retrieve images. On line 2 we import the *reports* and *report_images* services so that we can work directly with them. Line 6 retrieves that CAS Activity report, a system-monitoring report included in all SAS Viya environments (SAS Institute Inc. 2019c). The object returned by the service contains basic information about the report as well as metadata about the contents of each page in the report. Lines 8 and 9

filter those contents and isolate the “I/O and Threads” graph. The call on line 11 retrieves that content from the report.

Because web browsers are the primary client for these services, we can request the images at specific sizes and levels of detail, and the results will be rendered on the fly and returned as an SVG image. Since we didn’t specify a size or detail level, sasctl automatically uses reasonable defaults. Figure 1 shows the resulting visualization. Note that the result from line 11 is one or more SVG images, a standard format for web-based content, but there are common Python packages available to convert these to traditional raster formats (pyrsvg 2016).

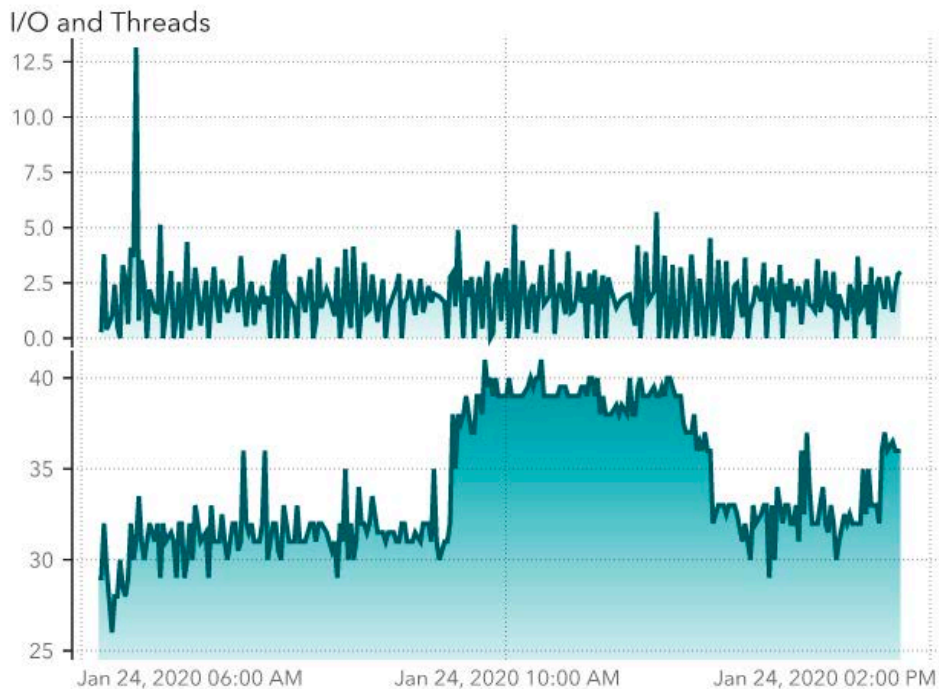


Figure 1. CAS Server Activity

LOW-LEVEL USAGE

Previous examples demonstrated how sasctl aims to be simple and easy to use, providing high-level interfaces for common tasks and simple service-level interfaces. As such, we haven’t focused on *what* is being sent to and returned from the SAS services when calling them. However, we understand that some users will want or need to have more control over their interactions with the SAS environment. The following example shows some of the lower-level ways to interact with SAS.

```
1 import pickle
2 from sasctl import get, get_link, request_link, Session
3
4 s = Session('example.sas.com', 'brian', 'N@ughtiusMax1mus')
5
6 response = get('files')
7
8 for link in response.links:
9     print(link)
```

In addition to the familiar Session object, line 2 imports a few new low-level functions. The first, `get()`, is used on line 6. This makes an HTTP GET request to the specified URL using the current session. In this example, the request call is to <https://example.sas.com/files>. This URL corresponds to the top-level URL for the Files service in a standard SAS environment, and the result is a dictionary representation of the REST response object (generally a JSON payload). This response can be used like a standard Python dictionary, or accessed using dot notation, similar to a Pandas DataFrame.

Many of the SAS microservices follow the HATEOAS paradigm (HATEOAS Driven REST APIs n.d.), and the standard is for services to return a `links` collection containing valid operations. Lines 8 and 9 iterate over this collection and display the available links.

```
{'method': 'HEAD', 'rel': 'checkState', 'href': '/files/files',
 'uri': '/files/files', 'type': 'application/json'}
{'method': 'POST', 'rel': 'create', 'href': '/files/files', 'uri': '/files/files',
 'type': '*/*', 'responseType': 'application/vnd.sas.file'}
{'method': 'GET', 'rel': 'files', 'href': '/files/files', 'uri': '/files/files',
 'type': 'application/vnd.sas.collection'}
{'method': 'POST', 'rel': 'bulkFiles', 'href': '/files/files',
 'uri': '/files/files', 'type': 'application/vnd.sas.selection',
 'responseType': 'application/vnd.sas.collection'}
```

Output 3. Available Links from the Top-level /files URL

Some response objects might have numerous valid operations, and therefore many different links available. If the name (`rel`) of the desired link is known, then the `get_link()` function can be used to retrieve the link information from the response.

```
10 get_link(response, 'files')
```

```
{'method': 'GET', 'rel': 'files', 'href': '/files/files', 'uri': '/files/files',
 'type': 'application/vnd.sas.collection'}
```

Output 4. The “Files” Link

While this makes it easy to get the information for a particular link, generally the goal is to actually make a request to that link. We use the `request_link()` function to make this call.

```
11 all_files = request_link(response, 'files')
12
13 for file in filter(lambda x: x.name == 'traincode.sas', all_files):
14     print(file)
```

Line 11 makes the request described in Output 4 and returns the results. In this case, that link retrieves the metadata about all of the files in the SAS environment (SAS Institute Inc. n.d.b). Since this is likely to be a very large list, the Files service supports pagination and returns only the first few results. However, `sasctl` automatically recognizes when this occurs and converts the response into a `PagedList` data structure. The `all_files` variable references such a data structure and operates just like a standard Python list but will transparently fetch data from the server only when needed.

Lines 13-14 demonstrate this capability by iterating through each file and filtering out those where the file name is `traincode.sas`. The (truncated) results are shown in Output 5. Note that even though we're only iterating through each file's metadata and not the actual file contents, this is still not a recommended practice as there may be thousands of files in the environment and `sasctl` will be forced to download the metadata for all of them.

```
traincode.sas
traincode.sas
traincode.sas
traincode.sas
...
```

Output 5. Client-Filtered Files Named “traincode.sas” (Truncated)

Instead, the recommended alternative is to use server-side filtering whenever possible, especially when dealing with potentially large collections. Most SAS services support multiple filtering methods (SAS Institute Inc. n.d.c) and since `sasctl` is built on the requests module (Reitz 2016) it is simple to pass additional parameters to `request_link()` to customize the request sent.

```
15 all_files = request_link(response, 'files', params={'filter': 'eq(name, traincode.sas)})
16 file = all_files[0]
17 content = request_link(file, 'content')
18 print(content)
```

Line 15 again retrieves a list of files named `traincode.sas`, but unlike before, it uses server-side filtering to only return the matching files to the client. Lines 16-18 select the first matching file and retrieve the actual content of the file. Output 6 shows the first few lines of that content, which in this case is SAS code.

```
*-----*;
* Macro Variables for input, output data and files;
  %let dm_datalib =;
  %let dm_lib      = WORK;
  %let dm_folder  = %sysfunc(pathname(work));
*-----*;
*-----*;
  * Training for tree;
*-----*;
*-----*;
  * Initializing Variable Macros;
*-----*;
```

```
%macro dm_unary_input;
%mend dm_unary_input;
%global dm_num_unary_input;
...
```

Output 6. Content of the traincode.sas File

In some cases, the file content might not be simple text, or we might want more control over how the response is handled. In that case, we can tell the `request_link()` function how to format the response. The following code snippet builds off the previous Scikit-Learn Model example and assumes that those files are present in the environment:

```
19 file = request_link(response, 'files', params={'filter': 'eq(name, "model.pkl")'})
20
21.pkl = request_link(file, 'content', format='content')
22
23 pickle.loads(pk1)
```

Line 19 requests the file called `model.pkl` from the SAS environment using another server-side filter. This is the file containing the pickled scikit-learn model that was automatically created by `sasctl` when the model was registered. In this case, we're assuming there's only one such file in the environment. If you have registered multiple such models in your environment, you might need to apply additional filtering.

Line 21 requests the actual content of the file. Since we know the file contains a binary pickle object and not regular text, we use the `format=` parameter to specify that we want the raw file contents returned instead of trying to parse it into text/JSON as is the default. The result is a binary string we unpickle on Line 23, giving us back the original scikit-learn model.

```
GradientBoostingRegressor(alpha=0.9, ccp_alpha=0.0, criterion='friedman_mse',
                           init=None, learning_rate=0.1, loss='ls', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=None, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0, warm_start=False)
```

Output 7. Unpickled scikit-learn Model from SAS Model Manager

CONCLUSION

We've demonstrated how the new `sasctl` package enables Python developers to integrate with the SAS platform without having to focus on the technical details of the integration. The single overriding goal is to make integration easy by providing the following:

- high-level operations for accomplishing common tasks.
- medium-level access to each SAS microservice for easy integration with specific services.

- low-level access to the underlying REST framework, allowing custom requests without having to worry about authentication, logging, or security.
- an easy way to retrieve all REST responses and requests, enabling the foundational REST interactions to be easily replicated in other tools and programming languages.

The `sasctl` package is intended to be a community-driven package as we believe the Python user community is best equipped to identify what functionality should be added or improved. As such, we welcome and greatly appreciate any contributions or feedback! The current version of `sasctl` contains many enhancements since its initial release in 2019, and we will continue to improve the package with input from our users.

APPENDIX

INSTALLING SASCTL

Install the `sasctl` package in any current Python environment using `pip`:

```
pip install sasctl
```

`sasctl` requires a few additional packages, but if these packages are not already present, they will be downloaded and installed automatically:

- `requests`
- `six`

Further, note that the examples described in this paper require functionality from some additional packages:

- `pandas`
- `sklearn`
- `swat`

ESTABLISHING SESSIONS

The first step in using `sasctl` is to establish a session to a SAS Viya server. When creating the session, `sasctl` performs a few steps behind the scenes:

- verifying the identity of the SAS server
- authenticating the user
- obtaining an authorization token

While the steps above are usually transparent to the user, it is important to understand these steps since establishing a session can sometimes cause difficulty for new users. By default, `sasctl` communicates with the SAS server using an encrypted HTTPS connection, and before establishing this connection it verifies the server's identity by validating the server's digital certificate. Generally, this is not a problem in production environments, but development and test environments often use servers with self-signed certificates that are not automatically trusted by your machine. If this is the case, you must either update your machine to trust the certificate or tell `sasctl` to skip the certificate verification step. There are a few different ways to do this (SAS Institute Inc. 2019e) but the easiest is usually to specify `verify_ssl=False` when creating the session, like the following:

```
s = Session('example.sas.com', 'arthur', 'K1ng0fTheBr!tons', verify_ssl=False)
```

REFERENCES

- Belsley, D. A., E. Kuh, and R. E. Welsch. 1980. *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. New York: Wiley. doi:10.1002/0471725153
- Mitchell, M. 2019. "Programming Languages for Data Scientists." Available <https://towardsdatascience.com/programming-languages-for-data-scientists-afde2eaf5cc5>.
- Pedregosa, Fabian, et al. 2011. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*. 12: 2825-2830. Available <http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf>.
- Reback, Jeff, et. al. 2019. "pandas-dev/pandas: v0.25.3." Available <http://doi.org/10.5281/zenodo.3524604>.
- Reitz, K. 2016. "Requests: HTTP for Humans™." Available <https://requests.readthedocs.io/en/master/> (accessed February 7, 2020).
- SAS Institute Inc. 2020a. SAS Software / python-sasctl. Available <https://github.com/sassoftware/python-sasctl/tree/master/examples> (accessed January 30, 2020).
- SAS Institute Inc. 2020b. SAS Software / python-swat. Available <https://github.com/sassoftware/python-swat> (accessed February 6, 2020).
- SAS Institute Inc. 2019a. "Concepts: Performance Monitoring," In SAS® *Model Manager 15.3: User's Guide*. Cary, NC: SAS Institute Inc. Available <https://go.documentation.sas.com/?cdcId=mdlmgrcdc&cdcVersion=15.3&docsetId=mdlmgrug&docsetTarget=p1c6xm7tthdajkn1t6esm4n3kwnq.htm> (accessed January 30, 2020).
- SAS Institute Inc. 2019b. "Model Publishing and Scoring Action Set: Syntax." In SAS® *Visual Analytics Programming Guide*. Cary, NC: SAS Institute Inc. Available https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=casapng&docsetTarget=cas-modelpublishing-runmodellocal.htm&locale=en (accessed February 1, 2020).
- SAS Institute Inc. 2019c. "Monitoring: How To (SAS Environment Manager)." In SAS® *Viya® 3.5 Administration: Monitoring*. Cary, NC: SAS Institute Inc. Available <https://go.documentation.sas.com/?cdcId=calcdc&cdcVersion=3.5&docsetId=calmonitoring&docsetTarget=n06ra7kjbev58n1lp0omm5jbjkwc.htm&locale=en#p0x0vy4rpruc20n1agvztukcsax0>.
- SAS Institute Inc. 2019d. "Python Support in SAS Micro Analytic Service." In SAS® *Micro Analytic Service 5.4: Programming and Administration Guide*. Cary, NC: SAS Institute Inc. Available https://go.documentation.sas.com/?docsetId=masag&docsetTarget=p1exphs802pzfgn1jlng_r4j4tmw0.htm&docsetVersion=5.4&locale=en (accessed January 30, 2020).
- SAS Institute Inc. 2019e. sasctl / Authentication. Available <https://sassoftware.github.io/python-sasctl/#authentication> (accessed January 30, 2020).
- SAS Institute Inc. 2019f. SAS® *Micro Analytic Service 5.4: Programming and Administration Guide*. Available <http://documentation.sas.com/?docsetId=masag&docsetVersion=5.4&docsetTarget=titlepage.htm> (accessed January 30, 2020).
- SAS Institute Inc. 2019g. SAS® *Model Manager 15.3: User's Guide*. Cary, NC: SAS Institute Inc. Available <https://go.documentation.sas.com/?cdcId=mdlmgrcdc&cdcVersion=15.3&docsetId=mdlmgrug&docsetTarget=titlepage.htm>.

SAS Institute Inc. 2018a. "Configuring Support for a DS2 PyMAS Package," In SAS® Micro Analytic Service 5.2: *Programming and Administration Guide*. Cary, NC: SAS Institute Inc. Available <https://go.documentation.sas.com/?docsetId=masag&docsetTarget=n1nznadmcs2hu6n1x9y8mmfvl0z3.htm&docsetVersion=5.2&locale=en> (accessed February 1, 2020).

SAS Institute Inc. 2018b. "DS2 Interface to Python." In SAS® Micro Analytic Service 5.2: *Programming and Administration Guide*. Cary, NC: SAS Institute Inc. Available <https://go.documentation.sas.com/?docsetId=masag&docsetTarget=n17lpph1l0p8xjn194yt4vv3o1sa.htm&docsetVersion=5.2> (accessed February 12, 2020).

SAS Institute Inc. n.d.a. SAS Viya REST APIs / Create a Publishing Destination. Available <https://developer.sas.com/apis/rest/DecisionManagement/#create-a-publishing-destination> (accessed February 12, 2020).

SAS Institute Inc. n.d.b. SAS Viya REST APIs / Files. Available <https://developer.sas.com/apis/rest/CoreServices/#get-file-resources> (accessed February 7, 2020).

SAS Institute Inc. n.d.c. SAS REST APIs: Filtering. Available <https://developer.sas.com/reference/filtering/> (accessed February 7, 2020).

SAS Institute Inc. n.d.d. SAS Viya REST APIs / Report Images. Available <https://developer.sas.com/apis/rest/Visualization/#report-images>.

SAS Institute Inc. n.d.e. SAS Viya Rest APIs / Reports. Available <https://developer.sas.com/apis/rest/Visualization/#reports>.

Smith, Kevin D. and Xiangxiang Meng. 2017. *SAS® Viya®: The Python Perspective*. Cary, NC: SAS Institute Inc.

"pyrsvg." cairo. 2016. Available <https://www.cairographics.org/cookbook/pyrsvg/> (accessed January 30, 2020).

"HATEOAS Driven REST APIs." Available <https://restfulapi.net/hateoas/> (accessed February 7, 2020).

ACKNOWLEDGMENTS

I want to thank Joe Furbee, Natalie Janes, and Sudhir Nallagangu for their contributions to this paper. And although they're too numerous to list here, I'd like to express my gratitude to everyone who has helped make sasctl what it is today by contributing code, ideas, bug reports, and moral support.

CONTACT INFORMATION

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

Jon Walker
SAS Institute Inc.
jonathan.walker@sas.com

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

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