Paper SAS3489-2019

Cows or Chickens: How You Can Make Your Models into Containers

Hongjie Xin, Jacky Jia, David Duling, Chris Toth

SAS Institute Inc.

ABSTRACT

Models are specific units of work that have one job to perform: scoring new data to make predictions. Containers are self-contained workers that can be easily created, destroyed, and reused as needed. They are portable and easily integrate into numerous modern cloud and on-premises execution engines. SAS® users can now follow a recipe to turn advanced model functions into on-demand containers such as Docker for both on-premises and cloud deployment. SAS® Model Manager can be used to organize the model content from many sources, including SAS and open source, to create containers. This presentation presents the basics and shows you how to turn your SAS analytical models into modern containers.

INTRODUCTION

THE ANALYTICAL LIFE CYCLE

Figure 1 illustrates the analytical life cycle.

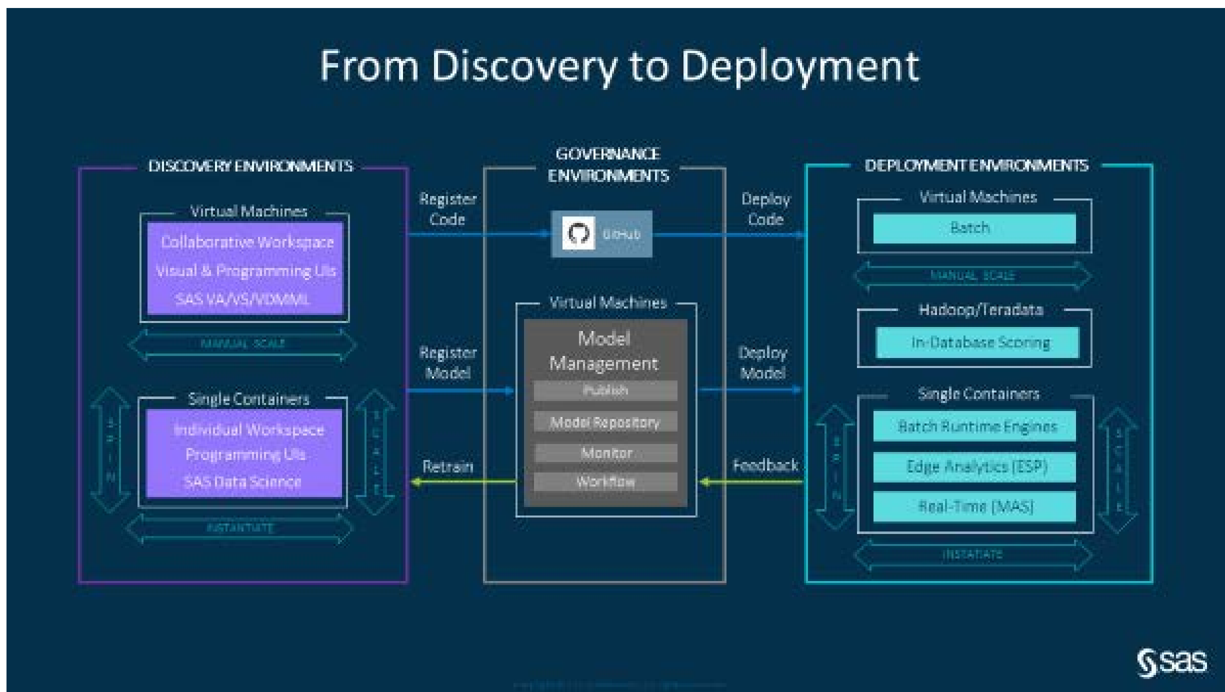

Figure 1. Analytical Lifecycle

Discovery environments have encompassed data mining and model training activities in collaborative workspaces, historically using virtual machines that are manually scaled for the expected workloads. Today, on-demand use cases are becoming more popular. Individual workspaces started with locally installed PCs. Data governance initiatives have evolved the architecture to use containers to provide governed access to data and code, as opposed to propagating multiple copies of data and code. Containers provide infrastructure

and compute usage cost savings, while providing a responsive user experience for the data scientist.

Organizations are exponentially increasing the number of models that are being built, due to their digital transformations. Machine learning and automation have lowered the costs to build a model. However, the cost of model governance has skyrocketed due to deployment and monitoring complexities. Model code developed in a Discovery environment will be registered in a Governance environment, such as a SAS Model Manager, or in a source code management system, such as GitHub.

The digital transformation has also necessitated the need for more robust execution options to deal with the explosion of data. Vast amounts of data need to be analytically enriched both at-rest in data lakes and enterprise data warehouses and in-flight in high-performance real-time business applications. Most organizations using SAS currently deploy their analytics using virtual machines and grids to manage disparate and on-demand workloads. Industry-leading organizations now use containers to facilitate on-demand and scalable processing for both batch and real-time workloads. Containers are providing similar infrastructure and compute usage cost savings as those experienced in a Discovery environment.

The feedback loop, providing operational results to the Governance environment, enables model performance monitoring and triggers automated model retraining in the Discovery environment. Operational data feedback closes the loop of the analytical life cycle.

A DAY IN THE LIFE OF A DATA SCIENTIST

A data scientist can spend weeks constructing a good model for prediction or classification using statistical, machine learning, or deep learning techniques. These models can be used to provide insight and inference into existing processes, or to predict outcomes based on new data values. These predictions are used to improve the effectiveness of automated decision making systems such as the next best offer, credit scoring, loan originations, fraud detection, robotic process automation, and hundreds of other applications. Modern businesses require the use of predictive models to remain competitive.

The building of predictive models is often termed *model training* and typically takes place offline in a development environment with saved historical data. The result of training a model is a fixed function that can be used for making predictions with new data values. The deployment of models is often termed *model scoring* and takes place in a production system running batch jobs or real-time recommendations. This step is where the model contributes real business value. However, there are several challenges in model deployment, as noted below:

- The discrepancy between model training systems and model scoring systems often results in the need to modify or completely rewrite model score code. This step is time consuming and requires expert staff resources.
- Delays in model deployment represent a loss of potential benefit derived from using the new model. This can have a large negative impact on the bottom line of the business.
- Model performance generally degrades over time as data values change with time and trends. Delays in deploying the model create delays in acquiring new data for model decay measurements. That period will delay training a new replacement model.
- The model must be deployed accurately. If the original trained model and the subsequent scoring model have even minor differences in floating-point values, missing value handling, or sequence of operations, errors can accumulate and create inaccuracies that will negatively impact model performance.

- The model deployment must be scalable. There are typically many models running in a production system. There can be multiple versions of the same model. Batch processes are scheduled to run in specific time periods or with constrained service level agreements. The load on real-time systems will vary by time of day, season, or external events, such as product discount sales.
- The model deployment must use standard information technology tools. The business's model production systems are often managed by staff that does not have experience with analytical tools. They are reluctant to add new processes every time the data scientists produce a new model. IT departments are also looking to reduce costs associated with maintaining too much hardware or acquiring upgraded hardware.

One remedy to these problems can be the use of modern light-weight containers. These devices are rapidly growing in popularity for systems and process management. The most popular container technology is Docker. A container is a compressed file that contains all the resources needed to execute a computational process. In this case we are creating containers to execute model scoring steps for both batch and real-time applications. The containers include the model score code and all the software that is needed to execute the model. This provides several benefits:

- The model does not need to be re-coded for different systems, eliminating several potential delays and errors.
- Model deployment can be much faster by standardizing the deployment process for any form of the model function.
- IT staff can use the same tools to manage model execution as any other IT-managed process, reducing staff training and expertise requirements.
- Multiple container instances provide a shared-nothing high availability. Failures in one instance will not affect other instances.
- New software releases can be added to new containers without affecting currently running systems.
- New models can be added to new containers without affecting currently running systems.
- Systems can be managed using standard container tools, such as Kubernetes. As demand increases, new container instances can be quickly created. As demand decreases, instances can be destroyed, freeing up resources for other tasks.

The traditional method of model deployment onto dedicated systems requires a large amount of resources and labor. The systems and processes must be carefully and expensively maintained. This is likened to owning a herd of cows. Each cow is precious and expensive. In contrast, containers are small replaceable units of labor. They can be quickly created and terminated. This is likened to a flock of chickens. Each chicken is disposable and cheap. Thus, the comparison can be represented as "cows versus chickens." (https://thenewstack.io/pets-and-cattle-symbolize-servers-so-what-does-that-make-containers-chickens/)

The remainder of this paper describes the details needed to turn both SAS models and open-source models into containers that can be treated as chickens. The paper uses the SAS® Viya® API to access model details and the Docker API to define and instantiate container instances. The result is a more scalable, more maintainable, and more efficient future.

DOCKER IMAGE OVERVIEW

A Docker image consists of multiple layers. Each of the layers is a read-only filesystem. The recipe for how to install the layers is defined in a Dockerfile. The last installed layer sits on top of the previous layers and hides the folders/files of previous layer if the folders/files have the exact same path. If a layer needs to modify the file in the lower layer, it first copies the file up to the target layer and then modifies it.

A container is an instance of the Docker image (from the docker run, docker create or Kubernetes commands). The Docker engine takes an image snapshot and adds a read/write filesystem on the top. It initializes the instance settings, such as IP address, system disk and memory resources, and so on.

To make bootup easier, the ENTRYPOINT statement in the Dockerfile could define an executable command after the instance has completed the initialization.

A Docker repository is a collection of different Docker images with same name but different tags. A tag is identified by an alphanumeric string. For example, semantic version number or build number is a common tag representation. The Docker registry is a service that hosts and distributes Docker images, such as Docker Hub and AWS/Google Container Registry. After the model image has been generated on the local host, we tag it and then push the tagged repository to a registry. Thus, the image could be referenced as format of an HTTP URL, for example, registryhost:5000/namespace/repo-name:tag.

MODEL IMAGE PUBLISHING

Transforming analytical models into containers is a very detailed and lengthy process. The remainder of this paper demonstrates how to publish a model image and test the model image with the Python utility library that SAS is developing.

Figure 2 shows the Python utility and its run-time environments

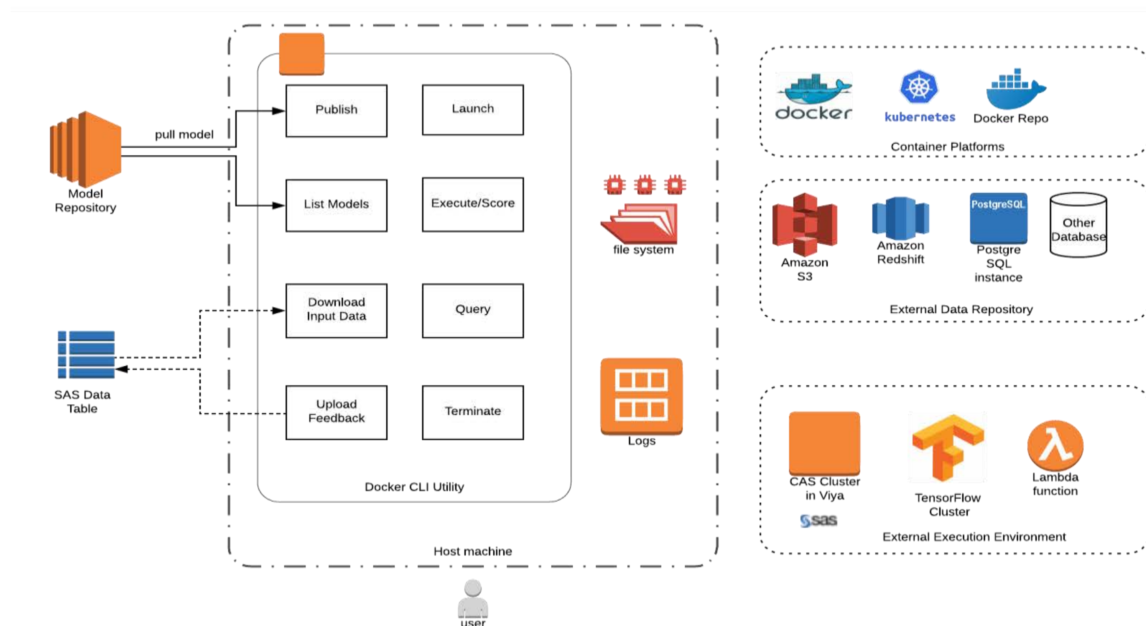

Figure 2. Python Utility

The model is stored in the SAS model repository. We can use the Python utility to pull the model ZIP file from the repository to the host machine. Then, we pack the model with the associated model base image and generate the model Docker image. After the model image is tagged with a version, the utility can push the image to the Docker repository and register it in the Docker registry.

We currently support three types of model base images (this might increase in the future):

- SAS® Micro Analytic Service (MAS) base image – to score SAS DS2 models
- PYML base image – to score Python models
- R base image – to score R models

Figure 3 shows the structure of the MAS base image.

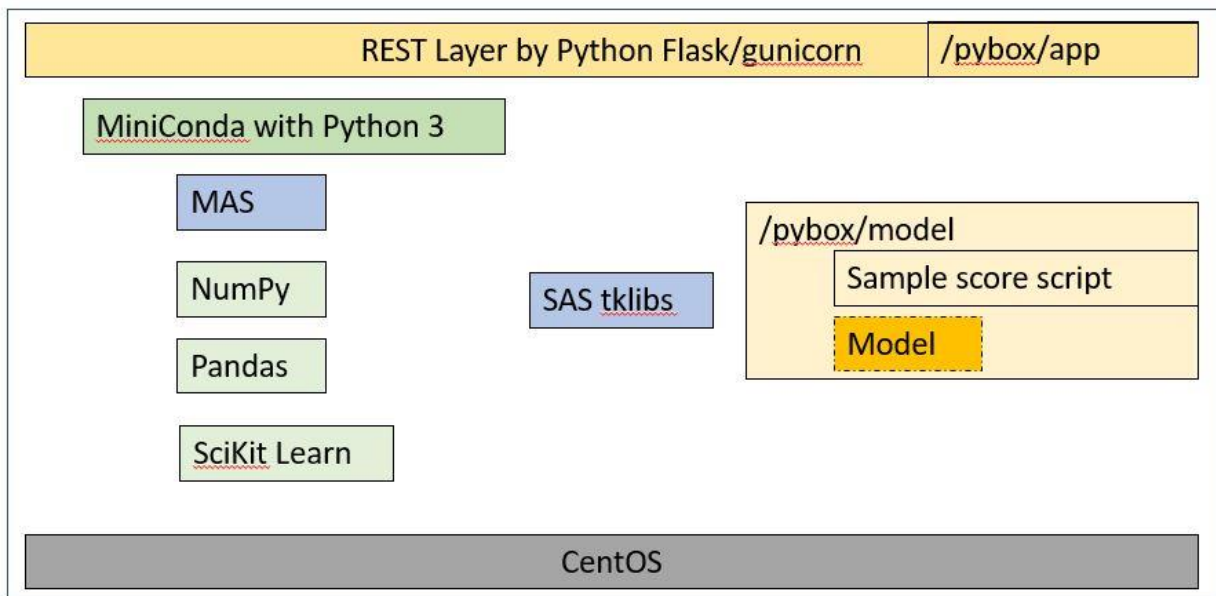

Figure 3. MAS Base Image Structure

The REST layer services web service calls from outside the container instance. In this base image we have included popular Python libraries and the MAS Python library under MiniConda as well as SAS threaded kernel (TK) libraries for MAS.

Figure 4 and Figure 5 show an example of querying model information, generating the model image, and pushing the image to the Docker repository.

```
In [1]: from mm_docker_lib import *
        initConfig()

In [2]: listmodel("svm")

        Model name svm (pipeline 1)
        Model UUID d00bb4e3-0672-4e9a-a877-39249d2a98ab
        Model version 63.0
        Project name MySVM
        Score Code Type ds2MultiType
        Image URL (not verified): docker.sas.com/honxin/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:la
        test
        ==========================
```

Figure 4. Jupyter Notebook - Execute the initConfig and listmodel Commands

```
In [3]: publish("d00bb4e3-0672-4e9a-a877-39249d2a98ab")

        Downloading model d00bb4e3-0672-4e9a-a877-39249d2a98ab from model repository...
        Copying astore from shared directory...
        Building image...
        Pushing to repo...
        Pushed. Please verify it at docker.sas.com/repository/honxin/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39
        249d2a98ab
        Model image URL: docker.sas.com/honxin/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest

Out[3]: 'docker.sas.com/honxin/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:63.0'
```

Figure 5. Jupyter Notebook – Execute the publish Command

MODEL VALIDATION

After the model image is generated and pushed to the Docker repository, users can launch the container instance at any time to score the model in the container instance. The launch command is shown in Figure 6.

```
In [4]: launch("docker.sas.com/honxin/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest")

        Launching container instance...
        Deployment created.
        Deployment name:  svm-pipeline-1-bgxzoo
        Service created.
        Getting service url...
        Service URL:  http://10.23.13.194:30847
        Checking whether the instance is up or not...
        Instance is up!

Out[4]: ('svm-pipeline-1-bgxzoo', 'http://10.23.13.194:30847')
```

Figure 6. Jupyter Notebook – Execute the launch Command

The launch command calls the Kubernetes API to create the deployment service object that exposes the deployment. Once the container instance is deployed, the service URL is available for scoring and querying.

MODEL SCORING

The initial version of the container REST API interface accepts only CSV as the input/output data format. Figure 7 shows the scoring and query test results.

```
In [5]: execute("http://10.23.13.194:30847", "hmeq.csv")

        Performing scoring in the container instance...
        The test_id from score execution: 1549592622.0851061

Out[5]: '1549592622.0851061'

In [6]: query(service_url="http://10.23.13.194:30847",test_id="1549592622.0851061")

        The test result has been retrieved and written into file 1549592622.0851061.csv
        Head is the first 5 lines
        EM_CLASSIFICATION,EM_EVENTPROBABILITY,EM_PROBABILITY,I_BAD,P_BAD0,P_BAD1,_P_,_WARN_
        0                              ,3.5255933e-05,0.9999648,0                                ,0.9999648,3.
        5255933e-05,1.0000224,
        0                              ,4.455951e-05,0.9999554,0                                ,0.9999554,4.4
        55951e-05,1.0000038,
        0                              ,3.431873e-05,0.99996567,0                               ,0.99996567,
        3.431873e-05,1.0000242,
        1                              ,1.0,1.0,1                                ,0.0,1.0,-1.1583366,

Out[6]: '1549592622.0851061.csv'
```

Figure 7. Jupyter Notebook - Scoring and Query Test Results

Figure 8 illustrates stopping a container instance and the related cleanup activities.

```
In [7]: stop(deployment_name="svm-pipeline-1-bgxzoo")

        Deleting service svm-pipeline-1-bgxzoo
        deleted svc/svm-pipeline-1-bgxzoo from ns/default
        Deleting app deployment... svm-pipeline-1-bgxzoo
        Deletion succeeded
```

Figure 8. Jupyter notebook – Execute the stop Command

As a best practice, stop a container when you are finished with your work. This minimizes infrastructure, compute usage, and related costs. The score command is a convenience command. It combines several commands that are commonly used together. Figure 9 shows the score command.

```
In [13]: score("docker.sas.com/honxin/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest", "hmeq.csv")

         Launching container instance...
         Deployment created.
         Deployment name:  svm-pipeline-1-5i99nr
         Service created.
         Getting service url...
         Service URL:  http://10.23.13.194:31725
         Checking whether the instance is up or not...
         Instance is up!
         ================================
         Performing scoring in the container instance...
         The test_id from score execution: 1549572368.693796
         ================================
         The test result has been retrieved and written into file 1549572368.693796.csv
         Head is the first 5 lines
         EM_CLASSIFICATION,EM_EVENTPROBABILITY,EM_PROBABILITY,I_BAD,P_BAD0,P_BAD1,_P_,_WARN_
         0                              ,3.5255933e-05,0.9999648,0                              ,0.9999648,3.
         5255933e-05,1.0000224,
         0                              ,4.455951e-05,0.9999554,0                              ,0.9999554,4.4
         55951e-05,1.0000038,
         0                              ,3.431873e-05,0.99996567,0                              ,0.99996567,
         3.431873e-05,1.0000242,
         1                              ,1.0,1.0,1                              ,0.0,1.0,-1.1583366,
         ================================
         Deleting service svm-pipeline-1-5i99nr
         deleted svc/svm-pipeline-1-5i99nr from ns/default
         Deleting app deployment... svm-pipeline-1-5i99nr
         Deletion succeeded
```

Figure 9. Jupyter Notebook – Execute the score Command

MODEL ASSESSMENT

The utility log and input/output data are organized in an SQLite file. Because the container life cycle could be very short, it is better to retrieve the score results from the container and store it in the host filesystem or an external database.

The SAS SWAT package is a Python interface to SAS® Cloud Analytic Services (CAS). With this package, you can load and analyze data sets of any size from your desktop or in the cloud. In addition, you can analyze extremely large data sets using as much processing power as you need, while still retaining the ease-of-use of Python on the client side.

Next, we can load the scoring output data into CAS for further analysis, for example, to **assess the model's performance.**

Figure 10 shows the loading of CSV data and using the SAS SWAT package to upload the test results into CAS.

```
In [4]: import swat
        import os
```

```
In [5]: os.environ["CAS_CLIENT_SSL_CA_LIST"] = "/opt/sas/viya/config/etc/SASSecurityCertificateFramework/cacerts/vault-ca.crt"
        cashost = 'summer.edmt.sashq-d.openstack.sas.com'
        casport = 5570
        casuser = 'edmdev'
        mycas = swat.CAS(cashost,casport,casuser,'Go4thsas')
```

```
In [57]: out1 = mycas.upload('hmeq_out.csv',
                             casout=dict(caslib='public',name='hmeq_out',replace=True))

NOTE: Cloud Analytic Services made the uploaded file available as table HMEQ_OUT in caslib public.
NOTE: The table HMEQ_OUT has been created in caslib public from binary data uploaded to Cloud Analytic Services.
```

```
In [58]: outTable = out1.casTable
```

```
In [59]: outTable.head()
```

Out[59]:

Selected Rows from Table HMEQ_OUT

	BAD	LOAN	MORTDUE	VALUE	REASON	JOB	YOJ	DEROG	DELINQ	CLAGE	NINQ	CLNO	DEBTINC	P_BAD0	P_BAD1
0	1.0	1100.0	25860.0	39025.0	HomeImp	Other	10.5	0.0	0.0	94.366667	1.0	9.0	NaN	0.000000	1.000000
1	1.0	1300.0	70053.0	68400.0	HomeImp	Other	7.0	0.0	2.0	121.833333	0.0	14.0	NaN	0.545455	0.454545
2	1.0	1500.0	13500.0	16700.0	HomeImp	Other	4.0	0.0	0.0	149.466667	1.0	10.0	NaN	0.000000	1.000000
3	1.0	1500.0	NaN	NaN			NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.636364	0.363636
4	0.0	1700.0	97800.0	112000.0	HomeImp	Office	3.0	0.0	0.0	93.333333	0.0	14.0	NaN	0.133333	0.866667

Figure 10. Using the SAS Swat Package to Upload Test Results into CAS

Figure 11 shows the assessment of the model.

```
In [60]: mycas.loadActionSet("percentile")

NOTE: Added action set 'percentile'.
```

Out[60]: **§ actionset**

percentile

elapsed 0.000479s · sys 0.000167s · mem 0.194MB

```
In [61]: r = outTable.percentile.assess(
               casOut=dict(name='assess',caslib='public',replace=True),
               rocOut=dict(name='assess_roc',caslib='public',replace=True),
               fitStatOut=dict(name='assess_fitstat',caslib='public',replace=True),
               cutStep=0.01,
               nBins=20,
               maxIters=50,
               inputs='p_bad1',
               response='BAD',
               event='1',
               pVar='p_bad0',
               pEvent='0'
             )
         r
```

Out[61]: **§ OutputCasTables**

	casLib	Name	Rows	Columns	casTable
0	Public	assess	20	21	CASTable('assess', caslib='Public')
1	Public	assess_roc	100	21	CASTable('assess_roc', caslib='Public')
2	Public	assess_fitstat	1	6	CASTable('assess_fitstat', caslib='Public')

elapsed 0.0257s · user 0.0244s · sys 0.00179s · mem 4.28MB

Figure 11. Assessing the Model

The next several figures are related to drawing plots.

Figure 12 shows the CAS table with lift.

```
In [68]: liftTable = r['OutputCasTables']['casTable'][0]
         liftTable.head(10)
Out[68]:
```

Selected Rows from Table ASSESS

	Column	_Event_	_Depth_	_Value_	_NObs_	_NEvents_	_NEventsBest_	_Resp_	_RespBest_	_Lift_	...	_CumResp_	_CumRespBe:
0	P_BAD1	1	5.0	1.000000	298.0	291.175573	298.0	24.489115	25.063078	4.897823	...	24.489115	25.063078
1	P_BAD1	1	10.0	0.800000	298.0	261.801700	298.0	22.018646	25.063078	4.403729	...	46.507761	50.126156
2	P_BAD1	1	15.0	0.615385	298.0	204.578283	298.0	17.205911	25.063078	3.441182	...	63.713672	75.189235
3	P_BAD1	1	20.0	0.428571	298.0	150.371274	295.0	12.646869	24.810765	2.529374	...	76.360541	100.000000
4	P_BAD1	1	25.0	0.312500	298.0	110.170732	0.0	9.265831	0.000000	1.853166	...	85.626372	100.000000
5	P_BAD1	1	30.0	0.200000	298.0	67.695542	0.0	5.693485	0.000000	1.138697	...	91.319857	100.000000
6	P_BAD1	1	35.0	0.100000	298.0	40.684508	0.0	3.421742	0.000000	0.684348	...	94.741599	100.000000
7	P_BAD1	1	40.0	0.000000	298.0	24.438281	0.0	2.055364	0.000000	0.411073	...	96.796963	100.000000
8	P_BAD1	1	45.0	0.000000	298.0	3.173676	0.0	0.266920	0.000000	0.053384	...	97.063883	100.000000
9	P_BAD1	1	50.0	0.000000	298.0	3.173676	0.0	0.266920	0.000000	0.053384	...	97.330803	100.000000

10 rows × 21 columns

Figure 12. Generate CAS Table – Lift

Figure 13 shows a lift chart.

```
In [79]: from bokeh.io import output_notebook, show
         from bokeh.layouts import gridplot
         from bokeh.plotting import figure
         TOOLS = "pan,wheel_zoom,box_zoom,reset,save,box_select"
         p1 = figure(title="Lift Chart", x_axis_label = "Depth",y_axis_label = "Cumulative Lift",tools=TOOLS)

         p1.line(liftTable['_Depth_'],liftTable['_CumLift_'],legend="my model", line_color="green",line_width = 2)
         output_notebook()
         show(gridplot([p1], ncols=1, plot_width=400, plot_height=400))
```

Loading BokehJS ...

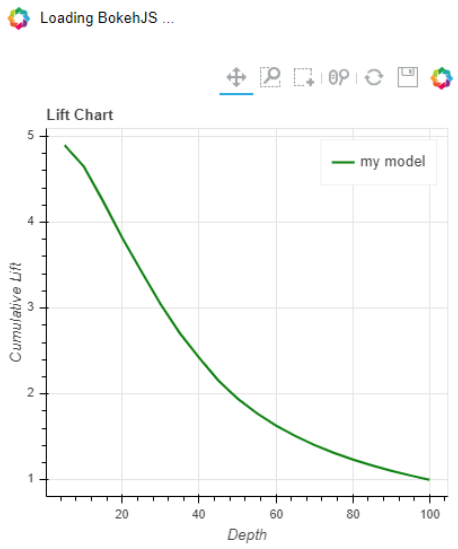

Figure 13. Generate Lift Chart

Figure 14 shows a CAS ROC table.

In [72]:
```
rocTable = r['OutputCasTables']['casTable'][1]
rocTable.head(10)
```

Out[72]:

Selected Rows from Table ASSESS_ROC

	Column	_Event_	_Cutoff_	_TP_	_FP_	_FN_	_TN_	_Sensitivity_	_Specificity_	_KS_	...	_FHALF_	_FPR_	_ACC_	_FDR_	_
0	P_BAD1	1	0.00	1189.0	4771.0	0.0	0.0	1.000000	0.000000	0.0	...	0.237524	1.000000	0.199497	0.800503	0.332
1	P_BAD1	1	0.01	1150.0	1148.0	39.0	3623.0	0.967199	0.759380	0.0	...	0.553897	0.240620	0.800839	0.499565	0.65
2	P_BAD1	1	0.02	1150.0	1148.0	39.0	3623.0	0.967199	0.759380	0.0	...	0.553897	0.240620	0.800839	0.499565	0.65
3	P_BAD1	1	0.03	1150.0	1148.0	39.0	3623.0	0.967199	0.759380	0.0	...	0.553897	0.240620	0.800839	0.499565	0.65
4	P_BAD1	1	0.04	1150.0	1148.0	39.0	3623.0	0.967199	0.759380	0.0	...	0.553897	0.240620	0.800839	0.499565	0.65
5	P_BAD1	1	0.05	1150.0	1148.0	39.0	3623.0	0.967199	0.759380	0.0	...	0.553897	0.240620	0.800839	0.499565	0.65
6	P_BAD1	1	0.06	1149.0	1126.0	40.0	3645.0	0.966358	0.763991	0.0	...	0.558363	0.236009	0.804362	0.494945	0.66
7	P_BAD1	1	0.07	1146.0	1089.0	43.0	3682.0	0.963835	0.771746	0.0	...	0.565702	0.228254	0.810067	0.487248	0.66
8	P_BAD1	1	0.08	1143.0	1053.0	46.0	3718.0	0.961312	0.779292	0.0	...	0.573047	0.220708	0.815604	0.479508	0.67
9	P_BAD1	1	0.09	1133.0	1000.0	56.0	3771.0	0.952902	0.790400	0.0	...	0.582759	0.209600	0.822819	0.468823	0.682

10 rows × 21 columns

Figure 14. Generate CAS Table – ROC

Figure 15 shows a ROC chart.

In [74]:
```
p2 = figure(title="ROC Chart",  x_axis_label = "1 - Specificity",y_axis_label = "Sensitivity",tools=TOOLS)
p2.line(rocTable['_FPR_'],rocTable['_Sensitivity_'],legend="my model", line_color="green",line_width = 2)

output_notebook()
show(gridplot([p2], ncols=1, plot_width=500, plot_height=500))
```

BokehJS 0.12.16 successfully loaded.

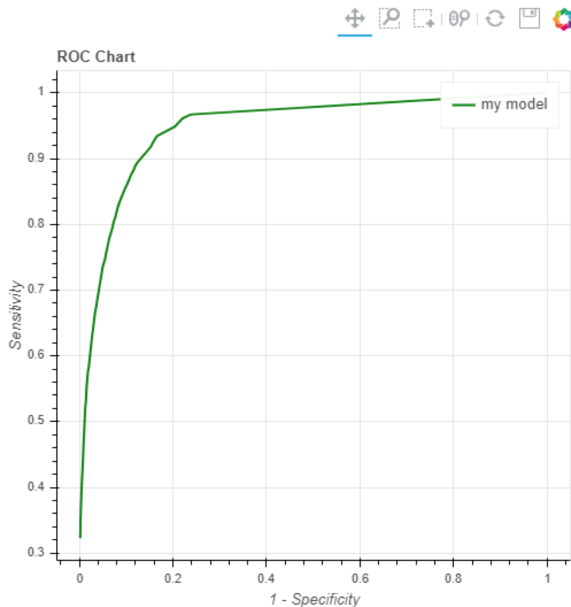

Figure 15. Generate ROC chart

We can also call the compare function to assess the model with multiple scoring results. Figure 16 shows the comparison between two scoring results.

```
In [11]: # service_url ='http://10.104.86.87:32367'
         # testIDs = {'q1': '1551321847.2215843', 'q2': '1551321852.1563883'}
         testIDs = {'q1':test_id1,'q2':test_id2}
         print(testIDs)
         compare(service_url,testIDs)

{'q1': '1551323661.6782281', 'q2': '1551323666.4520206'}
q1 = logs/1551323661.6782281.csv
NOTE: Cloud Analytic Services made the uploaded file available as table _OUT_Q1 in caslib public.
NOTE: The table _OUT_Q1 has been created in caslib public from binary data uploaded to Cloud Analytic Services.
q2 = logs/1551323666.4520206.csv
NOTE: Cloud Analytic Services made the uploaded file available as table _OUT_Q2 in caslib public.
NOTE: The table _OUT_Q2 has been created in caslib public from binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'percentile'.
==========Lift table==========
{'q1': CASTable('q1_assess', caslib='Public'),
 'q2': CASTable('q2_assess', caslib='Public')}
==========ROC table==========
{'q1': CASTable('q1_assess_ROC', caslib='Public'),
 'q2': CASTable('q2_assess_ROC', caslib='Public')}
```

BokehJS 0.12.16 successfully loaded.

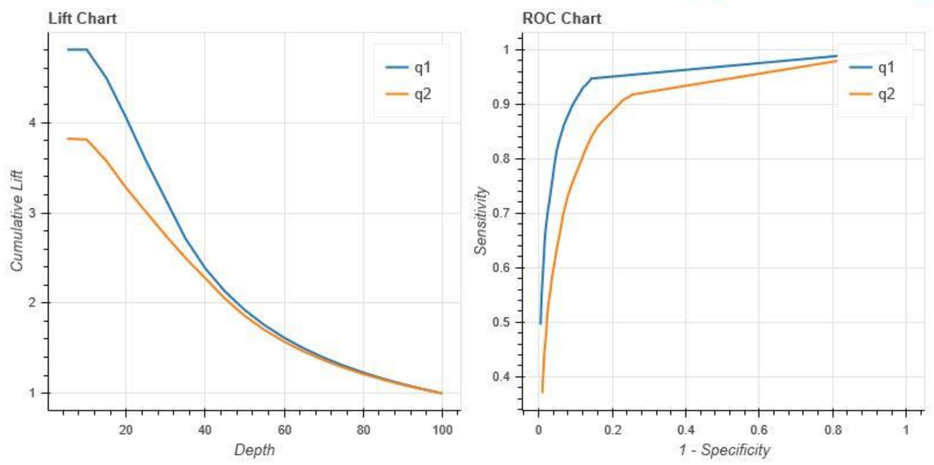

Figure 16. Side by Side Compare the Scoring Results

BEYOND THE MODEL

CLOUD

In addition to a private docker registry, we can upload a model image to a public docker registry, such as Docker Hub, Amazon Elastic Container Registry (ECR) or Google Container Registry (GCR), and then deploy the container instance in multiple cloud platforms.

Amazon Web Service (AWS)

Here is an example that illustrates how to register and store a model image in AWS and launch an AWS Elastic Container instance with Amazon Kubernetes.

First, we create and configure at least one Amazon Elastic Container Service for Kubernetes (EKS) cluster and its work nodes. This is shown in Figure 17.

	Stack Name	Created Time	Status	Drift Status	Description
	mm-docker-models-eks-worker-nodes	2019-02-14 10:08:09 UTC-0500	CREATE_COMPLETE	NOT_CHECKED	Amazon EKS - Node Group
	mm-docker-models-eks-vpc	2019-02-13 14:50:31 UTC-0500	CREATE_COMPLETE	NOT_CHECKED	Amazon EKS Sample VPC

Figure 17. AWS - CloudFormation for Kubernetes Cluster

Next, we set AWS properties in the cli.properties file. This is shown in Figure 18.

```
C:\test\SGF2019>type cli.properties
[CLI]
# set verbose mode, default is False
verbose=False

# run-time provider type. Available choices are: Dev, AWS, GCP
provider.type=AWS

[SAS]
# http to model repository
model.repo.host=http://honxin.modelmanager.sashq-d.openstack.sas.com

[Dev]
# set docker image url prefix, no http protocol
base.repo=docker.sas.com/honxin/

# set docker image repository web url prefix, no http protocol
base.repo.web.url=docker.sas.com/repository/honxin/

# kubernetes
kubernetes.context=minikube


[AWS]
# AWS config profile, copy one profile name from %USERPROFILE%\.aws\config
aws.profile=617292774228-sandbox

# this value will be automatically obtained from AWS ecr registry login
# base.repo=

# set docker image repository web url prefix, no http protocol
base.repo.web.url=console.aws.amazon.com/ecr/repositories/

# kubernetes
kubernetes.context=arn:aws:eks:us-east-1:617292774228:cluster/mm-docker-models-eks
```

Figure 18. Configure cli.properties File to Switch to AWS Cloud

By setting the provider type to AWS, the CLI utility publishes the model image to AWS ECR, and then deploys the model to an Amazon Elastic Container instance.

Figure 19 show the execution of initConfig and listmodel.

```
In [1]: from mm_docker_lib import *
        initConfig()

        Loading configuration properties...
        Login into AWS ECR...
          verbose: False
          model.repo.host: http://honxin.modelmanager.sashq-d.openstack.sas.com
          provider.type: AWS
          base.repo: 617292774228.dkr.ecr.us-east-1.amazonaws.com/
          base.repo.web.url: console.aws.amazon.com/ecr/repositories/
          kubernetes.context: arn:aws:eks:us-east-1:617292774228:cluster/mm-docker-models-eks
        ===================================

Out[1]: True

In [2]: listmodel("ds2")

        Model name ds2pkg_reg1_hmeq
        Model UUID c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3
        Model version 1.0
        Project name hmeq
        Score Code Type ds2Package
        Image URL (not verified): 617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-reg1-hmeq_c50e3b4f-e8ef-
        43d3-9e8d-7004f61e28b3:latest
        ==========================
```

Figure 19. AWS – Execute initConfig and listmodel Commands

Figure 20 shows the publishing of the model.

```
In [3]: publish("c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3")

        Downloading model c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3 from model repository...
        Building image...
        Creating remote repo ds2pkg-reg1-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3 in AWS ECR...
        Pushing to repo...
        Pushed. Please verify it at console.aws.amazon.com/ecr/repositories/ds2pkg-reg1-hmeq_c50e3b4f-e8ef-43d
        3-9e8d-7004f61e28b3/
        Model image URL: 617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-reg1-hmeq_c50e3b4f-e8ef-43d3-9e8d
        -7004f61e28b3:latest

Out[3]: '617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-reg1-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3:la
        test'
```

Figure 20. AWS – Execute publish Command

When the publish command is compete, the results can be verified in the AWS ECR Console. This is shown in Figure 21.

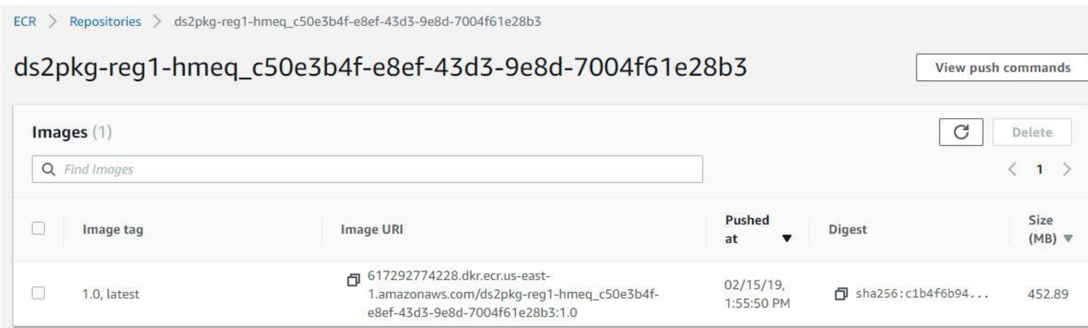

Figure 21. AWS – Use Elastic Container Registry (ECR) to Verify Results

Figure 22 shows the launching of the container instance in EKS.

```
In [4]: launch("617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-reg1-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e2

        Launching container instance...
        Deployment created.
        Deployment name:  ds2pkg-reg1-hmeq-s3f6gm
        Service created.
        Getting service url...
        Service URL:  http://54.87.222.138:30778
        Checking whether the instance is up or not...
        Instance is up!

Out[4]: ('ds2pkg-reg1-hmeq-s3f6gm', 'http://54.87.222.138:30778')
```

Figure 22. AWS - Launch Container Instance in EKS

Using the kubectl command line, we can verify information about the exposed service and the external IP address of the node. This is shown in Figure 23.

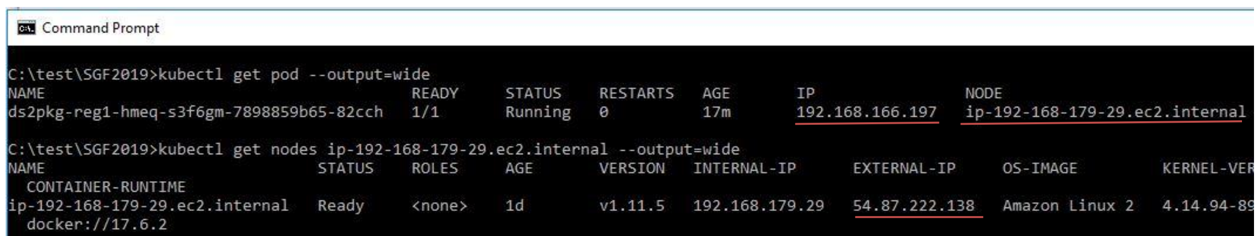

Figure 23. AWS – Verify Information with kubectl

Figure 24 shows scoring in an AWS container instance.

```
In [5]: execute("http://54.87.222.138:30778", "hmeq.csv")

        Performing scoring in the container instance...
        The test_id from score execution: 1550257389.6991549

Out[5]: '1550257389.6991549'
```

Figure 24. AWS - Perform Scoring in an AWS Container Instance

Figure 25 shows the execution of the query and stop commands.

```
In [6]: query(service_url="http://54.87.222.138:30778",test_id="1550257389.6991549")

        The test result has been retrieved and written into file 1550257389.6991549.csv
        Head is the first 5 lines
        EM_CLASSIFICATION,EM_EVENTPROBABILITY,EM_PROBABILITY,I_BAD,P_BAD0,P_BAD1,U_BAD,_WARN_
        0           ,0.075799346,0.92420065,0           ,0.92420065,0.075799346,0.0,None
        0           ,0.201893,0.798107,0           ,0.798107,0.201893,0.0,None
        0           ,0.05708384,0.94291615,0           ,0.94291615,0.05708384,0.0,None
        0           ,0.0974877,0.9025123,0           ,0.9025123,0.0974877,0.0,None

Out[6]: '1550257389.6991549.csv'

In [7]: stop(deployment_name="ds2pkg-reg1-hmeq-s3f6gm")

        Deleting service ds2pkg-reg1-hmeq-s3f6gm
        deleted svc/ds2pkg-reg1-hmeq-s3f6gm from ns/default
        Deleting app deployment... ds2pkg-reg1-hmeq-s3f6gm
        Deletion succeeded
```

Figure 25. AWS – Query Test Results and Delete the Deployment

Google Cloud Platform (GCP)

This section shows an example of deploying to Google Cloud Platform. The following images demonstrate how to push a model to Google Container Registration, how to launch a container instance in a Google Kubernetes cluster, and how to perform scoring and query results.

Figure 26 and Figure 27 show an example of executing the initConfig and listmodel commands, and then executing the publish command.

```
In [1]: from mm_docker_lib import *
        initConfig("GCP")

        Loading configuration properties...
        Login into GCP GCR...
        Login GCP GCR succeeded!
          verbose: False
          model.repo.host: http://honxin.modelmanager.sashq-d.openstack.sas.com
          provider.type: GCP
          base.repo: gcr.io/modelmanager/
          base.repo.web.url: console.cloud.google.com/gcr/images/modelmanager/GLOBAL/
          kubernetes.context: gke_modelmanager_us-east4-a_mm-docker-models-gke
        ===================================

Out[1]: True

In [2]: listmodel("svm")

        Model name svm (pipeline 1)
        Model UUID d00bb4e3-0672-4e9a-a877-39249d2a98ab
        Model version 63.0
        Project name MySVM
        Score Code Type ds2MultiType
        Image URL (not verified): gcr.io/modelmanager/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:late
        st
        ==========================
```

Figure 26. GCP – Execute initConfig and listmodel Commands

```
In [3]: publish("d00bb4e3-0672-4e9a-a877-39249d2a98ab")

        Downloading model d00bb4e3-0672-4e9a-a877-39249d2a98ab from model repository...
        Copying astore from shared directory...
        Building image...
        Pushing to repo...
        Pushed. Please verify it at console.cloud.google.com/gcr/images/modelmanager/GLOBAL/svm-pipeline-1_d00
        bb4e3-0672-4e9a-a877-39249d2a98ab/
        Model image URL: gcr.io/modelmanager/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest

Out[3]: 'gcr.io/modelmanager/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest'
```

Figure 27. GCP – Execute the publish Command

Figure 28 shows an example of using the Google Cloud Platform console to verify the results.

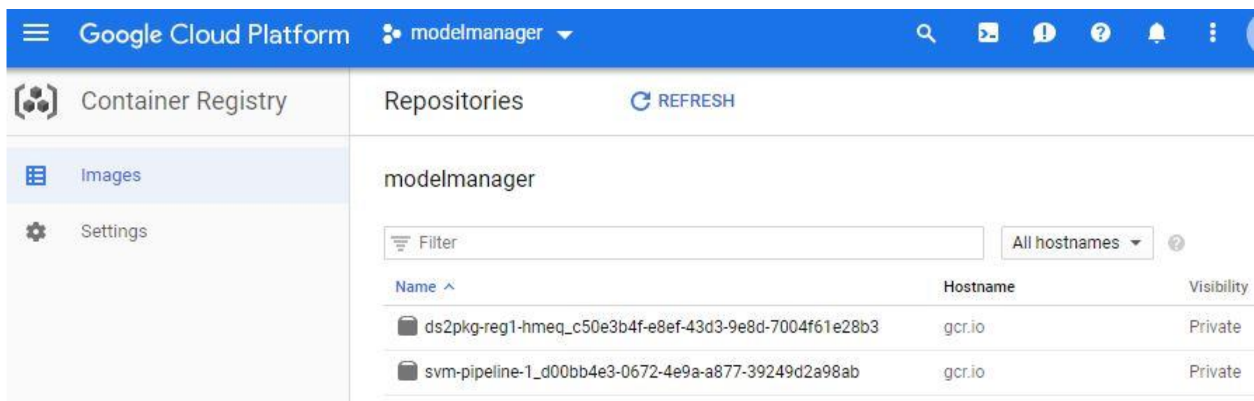

Figure 28. GCP – Use Google Cloud Platform Console to Verify Results

Figure 29 shows an example of executing the launch command.

```
In [4]: launch("gcr.io/modelmanager/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest")

        Launching container instance...
        Deployment created.
        Deployment name:  svm-pipeline-1-joe27k
        Service created.
        Getting service url...
        Service URL:  http://35.194.76.110:31480
        Checking whether the instance is up or not...
        1 ==Sleep 10 seconds...
        Checking whether the instance is up or not...
        Instance is up!

Out[4]: ('svm-pipeline-1-joe27k', 'http://35.194.76.110:31480')
```

Figure 29. GCP - Launch Container Instance in a Google Kubernetes Cluster

Figure 30 shows an example of verifying the deployment.

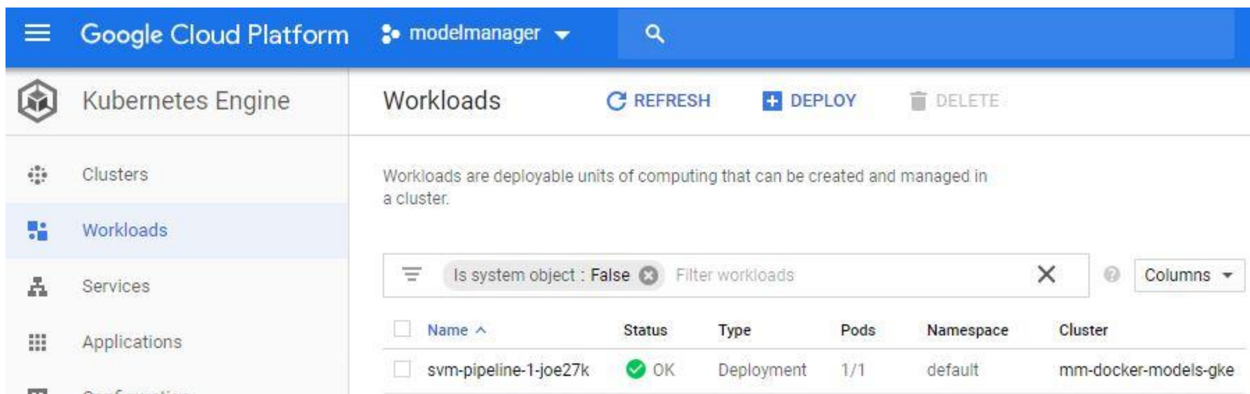

Figure 30. GCP – Verify Deployment in Google Kubernetes Engine Workloads

Figure 31 shows an example of verifying the service pod.

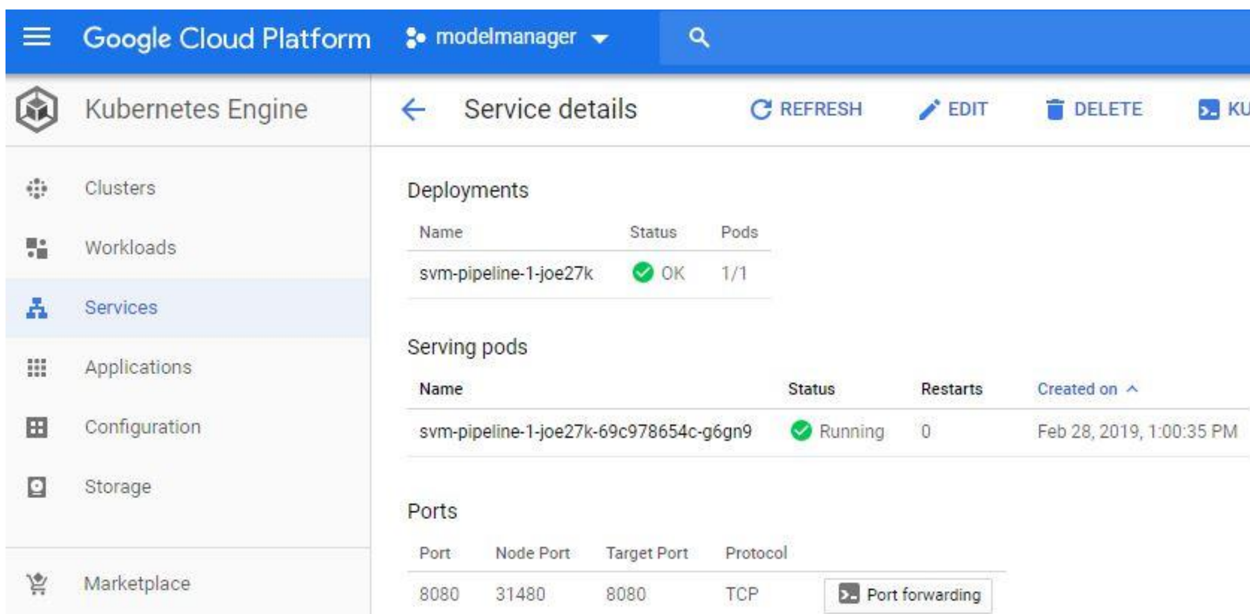

Figure 31. GCP – Verify Service Pod in Google Kubernetes Engine

Figure 32 shows scoring in an GKE container instance.

```
In [5]: execute("http://35.194.76.110:31480", "hmeq.csv")

        Performing scoring in the container instance...
        The test_id from score execution: 1551376925.196247

Out[5]: '1551376925.196247'
```

Figure 32. GCP - Perform Scoring in an GKE Container Instance

Figure 33 shows an example of querying the test results and deleting the deployment.

```
In [6]: query(service_url="http://35.194.76.110:31480",test_id="1551376925.196247")

        The test result has been retrieved and written into file 1551376925.196247.csv
        Head is the first 5 lines
        EM_CLASSIFICATION,EM_EVENTPROBABILITY,EM_PROBABILITY,I_BAD,P_BAD0,P_BAD1,_P_,_WARN_
        0                            ,3.5255933e-05,0.9999648,0                          ,0.9999648,3.
        5255933e-05,1.0000224,
        0                            ,4.455951e-05,0.9999554,0                          ,0.9999554,4.4
        55951e-05,1.0000038,
        0                            ,3.431873e-05,0.99996567,0                         ,0.99996567,
        3.431873e-05,1.0000242,
        1                            ,1.0,1.0,1                            ,0.0,1.0,-1.1583366,

Out[6]: '1551376925.196247.csv'

In [7]: stop(deployment_name="svm-pipeline-1-joe27k")

        Deleting service svm-pipeline-1-joe27k
        deleted svc/svm-pipeline-1-joe27k from ns/default
        Deleting app deployment... svm-pipeline-1-joe27k
        Deletion succeeded
```

Figure 33. GCP - Query Test Results and Delete the Deployment

DEPENDENCY SUPPORT

Our predefined base images could include the most popular libraries or packages. In the real world, a **user's model** might have extra dependencies on other software libraries or packages. Our solution to provide a mechanism to adapt to dynamic user requirements is as follows. The user:

1. Creates a file named requirements.json
2. Describes the steps about how to install extra dependencies in the file
3. Inserts this specification file in the model content list

When packing the model into the model image, the utility scans the specification file from model content list and includes those step commands as part of Dockerfile. The Dockerfile will be rendered by Docker Engine. For example, one data model is based on a Python H2O library that the base image has not packaged yet. This is illustrated in Figure 34.

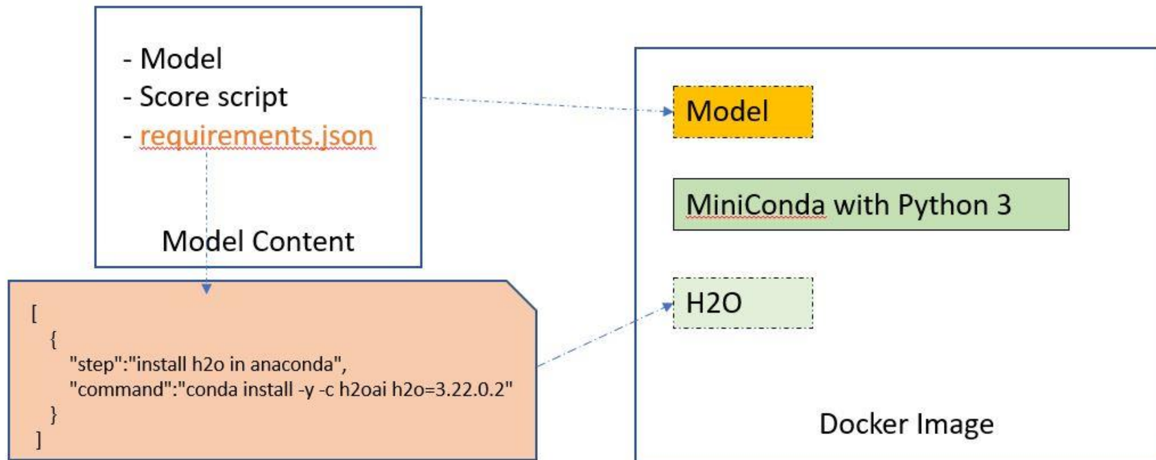

Figure 34. Support Extra Model Dependency

Figure 35 shows the specification file in the model content.

XGBoost (1.0) Publish Close

Files Variables Properties Versions

XGBoost_model_python_1547223300262_1
readme.txt
requirements.json
score.py
train.py

requirements.json (Read-Only)

```
1   [
2       {
3           "step":"install openjdk1.8 devel",
4           "command":"yum -y install java-1.8.0-openjdk-devel"
5       },
6       {
7           "step":"install h2o in anaconda",
8           "command":"conda install -y -c h2oai h2o=3.22.0.2"
9        }
10   ]
```

Figure 35. Specification File in the Model Content

Figure 36 shows the installation of the dependent packages in the image generation.

```
In [8]: setVerbose(True)
        publish("2fd01d3e-ac53-406d-86cd-ac3cc9557c57")

        Verbose: True
        Downloading model 2fd01d3e-ac53-406d-86cd-ac3cc9557c57 from model repository...
        ...
        Installing dependencies defined from requirements.json...
        Inserting dependency lines in Dockerfile
        #install openjdk1.8 devel
        RUN yum -y install java-1.8.0-openjdk-devel
        #install h2o in anaconda
        RUN conda install -y -c h2oai h2o=3.22.0.2

        Docker repository URL:  docker.sas.com/honxin/
        Building image...
        ...
        Model image URL: docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc9557c57:latest
        ==========================
        Guides: > python mm_docker_cli.py launch docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc95
        57c57:latest
        Guides: > python mm_docker_cli.py score docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc955
        7c57:latest <input file>

Out[8]: 'docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc9557c57:1.0'
```

Figure 36. Installing the Dependent Packages in Image Generation

When Verbose is set to True, the utility displays more useful output for each command. This is shown in Figure 37.

```
In [10]: setVerbose(True)
         launch("docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc9557c57:latest")

         Verbose: True
         Launching container instance...
         docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc9557c57:latest
         xgboost
         Deployment created.
         Deployment name:  xgboost-goprz9
         Service created.
         Getting service url...
         Service URL:  http://10.23.13.194:32634
         ==========================
         Checking whether the instance is up or not...
         Instance is up!
         Guides: > python mm_docker_cli.py execute  http://10.23.13.194:32634 <input file>
         Guides: > python mm_docker_cli.py stop xgboost-goprz9

Out[10]: ('xgboost-goprz9', 'http://10.23.13.194:32634')
```

Figure 37. Displaying More Information with Verbose Enabled

CONCLUSION

The goal of this paper is to show how to use our CLI utility library to pack a SAS or open-source model in a Docker image and perform scoring in a Docker container. It introduced the features of the current development stage of the CLI utility library. This paper might be updated in the future if we support more model types and additional cloud environments.

REFERENCES

Mouat, A. 2016. Using Docker: Developing and Deploying Software with Containers. 1st ed.: O'Reilly Media.

Docker Inc.. "Docker SDK for Python." Available at https://docker-py.readthedocs.io/en/stable/.

GitHub Inc.. "Python Kubernetes Client." Available at https://github.com/kubernetes-client/python/blob/master/kubernetes/README.md.

Amazon Web Services, Inc.. "Kubernetes AWS." Available at https://aws.amazon.com/kubernetes/.

Google Cloud. "Google Kubernetes Engine Documentation." Available at https://cloud.google.com/kubernetes-engine/docs/.

Bernard Golden, Mar 16 2015. "Pets and Cattle Symbolize Servers, so What Does That Make Containers? Chickens?" Available at https://thenewstack.io/pets-and-cattle-symbolize-servers-so-what-does-that-make-containers-chickens/.

RECOMMENDED READING

- *SAS® Model Manager 15.2: User's Guide*
- *SAS® Micro Analytic Service 5.2: Programming and Administration Guide*

CONTACT INFORMATION

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

David Duling
SAS Institute Inc.
919-677-8000
David.Duling@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.