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.

From Discovery to Deployment

![Diagram](image)

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.”

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.
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.

**Figure 4. Jupyter Notebook - Execute the initConfig and listmodel Commands**
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")
```

```
```

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", "hmeeq.csv")
```

```
Out[5]: '1549592622.0851061'
```

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

```
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-bgzxoo")

Deleting service svm-pipeline-1-bgzxoo
deleted svc/svm-pipeline-1-bgzxoo from ns/default
Deleting app deployment... svm-pipeline-1-bgzxoo
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_d00b4e3-0d72-4e9a-a877-39249d2a98ab:latest", "hmeq.csv")
```

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

Figure 11 shows the assessment of the model.

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)
```

<table>
<thead>
<tr>
<th>Column</th>
<th>Event</th>
<th>Depth</th>
<th>Value</th>
<th>NObs</th>
<th>NEEvents</th>
<th>NEEventsBest</th>
<th>Resp</th>
<th>RespBest</th>
<th>Lift</th>
<th>CumResp</th>
<th>CumRespBe</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>5.0</td>
<td>1.000000</td>
<td>259.0</td>
<td>291.175573</td>
<td>259.0</td>
<td>24.493115</td>
<td>25.063076</td>
<td>4.697623</td>
<td>24.493115</td>
<td>25.063076</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>10.0</td>
<td>0.800000</td>
<td>259.0</td>
<td>251.810700</td>
<td>259.0</td>
<td>22.016466</td>
<td>25.063076</td>
<td>4.403729</td>
<td>25.063076</td>
<td>4.403729</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>15.0</td>
<td>0.615865</td>
<td>259.0</td>
<td>254.578283</td>
<td>259.0</td>
<td>17.206911</td>
<td>25.063076</td>
<td>3.411982</td>
<td>63.713672</td>
<td>75.190235</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>20.0</td>
<td>0.420671</td>
<td>259.0</td>
<td>150.371274</td>
<td>259.0</td>
<td>12.548859</td>
<td>24.810766</td>
<td>2.529374</td>
<td>76.360641</td>
<td>100.000000</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>25.0</td>
<td>0.312500</td>
<td>259.0</td>
<td>110.179323</td>
<td>259.0</td>
<td>9.259631</td>
<td>0.000000</td>
<td>1.851195</td>
<td>85.526572</td>
<td>100.000000</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>30.0</td>
<td>0.200000</td>
<td>259.0</td>
<td>67.695542</td>
<td>259.0</td>
<td>5.693485</td>
<td>0.000000</td>
<td>1.135697</td>
<td>91.319657</td>
<td>100.000000</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>35.0</td>
<td>0.100000</td>
<td>259.0</td>
<td>40.684506</td>
<td>259.0</td>
<td>3.421742</td>
<td>0.000000</td>
<td>0.684548</td>
<td>54.741699</td>
<td>100.000000</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>40.0</td>
<td>0.000000</td>
<td>259.0</td>
<td>24.438281</td>
<td>259.0</td>
<td>2.055364</td>
<td>0.000000</td>
<td>0.411073</td>
<td>96.796963</td>
<td>100.000000</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>45.0</td>
<td>0.000000</td>
<td>259.0</td>
<td>3.173676</td>
<td>259.0</td>
<td>0.266920</td>
<td>0.000000</td>
<td>0.053384</td>
<td>57.036383</td>
<td>100.000000</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>1</td>
<td>50.0</td>
<td>0.000000</td>
<td>259.0</td>
<td>3.173676</td>
<td>259.0</td>
<td>0.266920</td>
<td>0.000000</td>
<td>0.053384</td>
<td>97.339603</td>
<td>100.000000</td>
</tr>
</tbody>
</table>

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,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["cum_lift"], legend="my model", line_color="green", line_width = 2)
output_notebook()
show(gridplot([p1, ncols=1, plot_width=400, plot_height=400]))
```

![Lift Chart](image)

Figure 13. Generate Lift Chart

Figure 14 shows a CAS ROC table.
Figure 14. Generate CAS Table – ROC

Figure 15 shows a ROC chart.

![ROC Chart](image)

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.
11

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.

<table>
<thead>
<tr>
<th>Stack Name</th>
<th>Created Time</th>
<th>Status</th>
<th>Dntt Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm-docker-models-eks-worker-nodes</td>
<td>2019-02-14 10:08:09 UTC-0000</td>
<td>CREATE_COMPLETE</td>
<td>NOT_CHECKED</td>
<td>Amazon EKS - Node Group</td>
</tr>
<tr>
<td>mm-docker-models-eks-vpc</td>
<td>2019-02-13 14:00:31 UTC-0000</td>
<td>CREATE_COMPLETE</td>
<td>NOT_CHECKED</td>
<td>Amazon EKS Sample VPC</td>
</tr>
</tbody>
</table>

Figure 17. AWS - CloudFormation for Kubernetes Cluster

Next, we set AWS properties in the cli.properties file. This is shown in Figure 18.
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.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(\r\n\"c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3\"
)
Downloading model c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3 from model repository...
Building image...
Creating remote repo ds2pkg-regl-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3 in AWS ECR...
Pushing to repo...
Pushed. Please verify it at console.aws.amazon.com/ecr/repositories/ds2pkg-regl-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3/
Model image URL: 617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-regl-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3:latest
```

Out[3]:

```
'617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-regl-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3:latest'
```

**Figure 20. AWS – Execute publish Command**

When the publish command is complete, 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(\r\n\r\n\"617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-regl-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3\"
)
```

Out[4]:

```
('ds2pkg-regl-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.

```python
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.

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM_CLASSIFICATION</td>
<td>0.075799345</td>
</tr>
<tr>
<td>EM_EVENTPROBABILITY</td>
<td>0.92420065</td>
</tr>
<tr>
<td>EM_PROBABILITY</td>
<td>0.38532165</td>
</tr>
<tr>
<td>I_BAD</td>
<td>0.291893</td>
</tr>
<tr>
<td>P_BAD0</td>
<td>0.798187</td>
</tr>
<tr>
<td>P_BAD1</td>
<td>0.50788384</td>
</tr>
<tr>
<td>U_BAD</td>
<td>None</td>
</tr>
</tbody>
</table>

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

Out[7]: None
```

**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.
Figure 26. GCP – Execute `initConfig` and `listmodel` Commands

```
In [1]:
    from mn_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.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 63.8
Project name MySVM
Score Code Type ds2MultiType
Image URL (not verified): gcr.io/modelmanager/svm-pipeline-1_d00bb4e3-0672-4e9a-a877-39249d2a98ab:latest
```

```
Out[2]:
```

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_d60bb4e3-8672-4e9a-a877-39246d2a98ab: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...
- 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.
Figure 32. GCP - Perform Scoring in an GKE Container Instance

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

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.

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 h2os1 h2o=3.21.0.2

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-ac3cc9557c57:latest
Guides: > python mm_docker_cli.py score docker.sas.com/honxin/xgboost_2fd01d3e-ac53-406d-86cd-ac3cc9557c57:latest <input file>
```

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

**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
```

```
```

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

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.