Paper SAS4402-2020

Open-Source Model Management with SAS® Model Manager

Glenn Clingroth, Hongjie Xin, and Scott Lindauer, SAS Institute Inc.

ABSTRACT

Open-source models that are developed in Python, R, TensorFlow, and so on, are increasingly important to organizations that produce and deploy analytical and machine learning models. Not only are the models created using open-source tools, they are deployed to open-source environments that use Docker and Kubernetes in place of more traditional environments. SAS® Model Manager is evolving to be a management platform that handles traditional SAS models and open-source models as equal partners. This paper discusses strategies for managing the life cycles of Python, R, and TensorFlow models using SAS Model Manager.

INTRODUCTION

In the open-source world, Python and R have become the prevalent analytic modeling languages. Packages such as scikit-learn and scipy provide powerful analytics, but the standard problem applies: how to take the model from development to production.

Since its inception, Model Manager has primarily worked with models produced using SAS. This includes models that are created in Enterprise Miner or Model Studio, or written in SAS® Data Step or using Proc invocations. However, Model Manager also works with models produced outside of SAS environments by importing PMML files. And since Model Manager can manage any set of files, it has always been possible to manage the code of models developed in the Python and R languages. But managing and versioning the code is not the same as managing the model.

As many organizations are discovering, data scientists are very good at producing models. These models, however, must be managed as part of the model life cycle. For example, models require:

- Vetting to make sure that the models are solving the problem
- Comparing to make sure that the organization selects the best model
- Deploying so that the models can do the work
- Monitoring to make sure that the models continue to do a good job

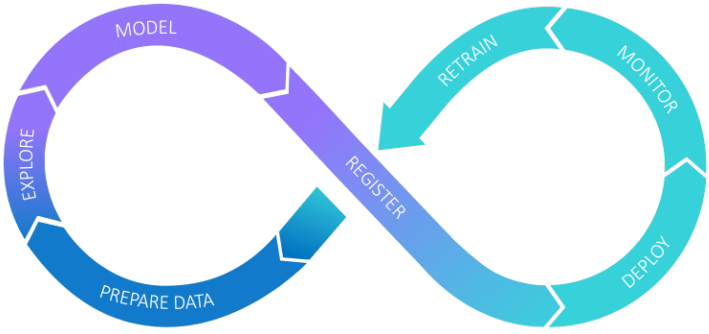


Figure 1: The model life cycle

Model Manager can play a large part in the model life cycle. But as stated previously, in the past it has been primarily a code repository for models that were not proprietary SAS models. A goal for the SAS® Viya® version of Model Manager is to treat open-source models as first-class models that can be part of the entire model life cycle. This means that they can be tested, deployed, and monitored either on their own or alongside SAS models. Model Manager has evolved to provide this functionality, and by providing the ability to publish models to standalone docker containers, it becomes possible to deploy the models into open-source environments.

Recently, SAS released SAS® Open Model Manager, which is targeted at the open-source community. Open Model Manager provides most of the features associated with standard Model Manager delivered in a standalone docker container. While Open Model Manager is not integrated with other SAS products, it provides a powerful platform for working with models written in the Python and R languages and treats those models as first-class models. All features described in this paper can be used in either Model Manager or Open Model Manager.

This paper provides a skeleton for using Model Manager with open-source models to perform the following actions:

- Registering open-source models into the model repository
- Comparing and validating the models prior to deployment
- Deploying open-source models to standalone containers
- Monitoring the model performance

All of the coding and model examples in this paper are written in Python.

WRITING CODE AND REGISTERING A MODEL

The first step in the model life cycle is to create the model code. This can come from hand writing the code, but increasingly modelers are using machine learning algorithms to discover their models.

Here is an example of a simple model discovery method in the scikit-learn package that generates a decision tree model:

```
from pathlib import Path
import pandas as pd
import sklearn.tree as tree
from sklearn.model_selection import train_test_split

dataFolder = Path.cwd() / 'hmeq/DATA'
zipFolder = Path.cwd() / 'hmeq/ZIP'
modelPrefix = 'hmeqClassTree'
yName = 'BAD'
catName = ['JOB', 'REASON']
intName = ['CLAGE', 'CLNO', 'DEBTINC', 'DELINQ', 'DEROG', 'NINQ', 'YOJ']
dataPath = (Path(dataFolder) / 'hmeq.csv')
inputData =
   pd.read_csv(dataPath, sep=',', usecols=[yName]+catName+intName)
useColumn = [yName]
useColumn.extend(catName + intName)
inputData = inputData[useColumn].dropna()
resultCol = inputData[yName]
xTrain, xTest, yTrain, yTest =
   train_test_split(inputData, resultCol, test_size=0.2, random_state=42)
model = tree.DecisionTreeClassifier(criterion='entropy', max_depth=5,
   min_samples_split=20, min_samples_leaf=10, random_state=42)
```

```
  DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=5,
     max_features=None, max_leaf_nodes=None,  min_impurity_decrease=0.0,
     min_impurity_split=None, min_samples_leaf=10, min_samples_split=20,
     min_weight_fraction_leaf=0.0, presort=False, random_state=42,
     splitter='best')
  x = pd.get_dummies(xTrain[catName].astype('category'))
  x = x.join(xTrain[intName])
  y = yTrain.astype('category')
  trainedModel = model.fit(x, y)
```

The trainedModel variable now contains a serialized model. The next step is to register the model with Model Manager. For Model Manager the model can be just the model code or the serialized model written to a pickle file, but at that point only the function of the model is being managed. To work well in a model life cycle, we also need some understanding of the model. For that we need to know the input and output variables used by the model. It is also useful to know something about the training data and how the model works with that data. To help with this, we provide the following package: pickeZip-mm.

You can use pickleZip-mm to get all of the metadata that Model Manager needs to work with the model, such as the input and output variables, model algorithm and model function. But to gain a better understanding of the model and how it compares to other models, pickleZip-mm also produces a collection of fit statistics as well as Lift and Roc charts.

The following example shows how to work with pickleZip-mm, the pzmm package, to register the newly trained model into Model Manager:

```
import pzmm
yCategory = y.cat.categories
outputVar = pd.DataFrame(columns=['EM_EVENTPROBABILITY','EM_CLASSIFICATION'])
outputVar['EM_CLASSIFICATION'] = yCategory.astype('str')
outputVar['EM_EVENTPROBABILITY'] = 0.5
inputVar = inputData[catName+intName]

modelName = 'Home Equity Loan Classification Tree'
remotePath = 'tmp/' + modelPrefix + '.pickle'
predictMethod = f'{model}.predict_proba({input})'

# Crate the score code for the model and write it to the file system
SC = pzmm.ScoreCode()
SC.writeScoreCode(inputDF=inputData[catName+intName],
                  targetDF=inputData[yName],
                  modelPrefix=modelPrefix,
                  predictMethod=predictMethod,
                  pRemotePath=remotePath,
                  pyPath=zipFolder)

# Write the pickle file to the file system
pzmm.PickleModel.pickleTrainedModel(trainedModel, modelPrefix, zipFolder)

# Write the JSON metadata files to the file system
JSONFiles = pzmm.JSONFiles
JSONFiles.writeVarJSON(inputVar, isInput=True, jPath=zipFolder)
JSONFiles.writeVarJSON(outputVar, isInput=False, jPath=zipFolder)

modelName = 'Home Equity Loan Classification Tree'
JSONFiles.writeModelPropertiesJSON(modelName=modelName,
```

```
                                    modelDesc='',
                                    targetVariable=yName,
                                    modelType='tree',
                                    modelPredictors=(catName + intName),

targetEvent=yCategory[1].astype('str'),
                                numTargetCategories=len(yCategory),
                                eventProbVar='EM_EVENTPROBABILITY',
                                jPath=zipFolder)

JSONFiles.writeFileMetadataJSON(modelPrefix, jPath=zipFolder)

# Calculate fit statistics, roc, and lift charts
trainData = inputData
validatePath = (Path(dataFolder) / 'hmeq_validate.csv')
validateData =
  pd.read_csv(validtePath, sep=',', usecols=[yName]+catName+intName)
testPath = (Path(dataFolder) / 'hmeq_test.csv')
testData =
  pd.read_csv(testPath, sep=',', usecols=[yName]+catName+intName)

calculateFitStats(validateData, trainData, testData, zipFolder)
calculateLiftStats(validateData, trainData, testData, zipFolder)
calculateROCStats(validateData, trainData, testData, zipFolder)

# Create a zip file
pzmm.ZipModel.zipFiles(fileDir=zipFolder, modelPrefix=modelPrefix)

# Import the model into Model Manager
host = 'modelmanager.mycompany.com'
ModelImport = pzmm.ModelImport(host)
zPath = Path(zipFolder) / (modelPrefix + '.zip')
ModelImport.importModel(modelPrefix, projectName='ML_HMEQ', zPath=zPath)
```

With the importModel call, the ZIP file is imported and a model named “hmeqClassTree” is imported into a Model Manager project named “ML_HMEQ”.

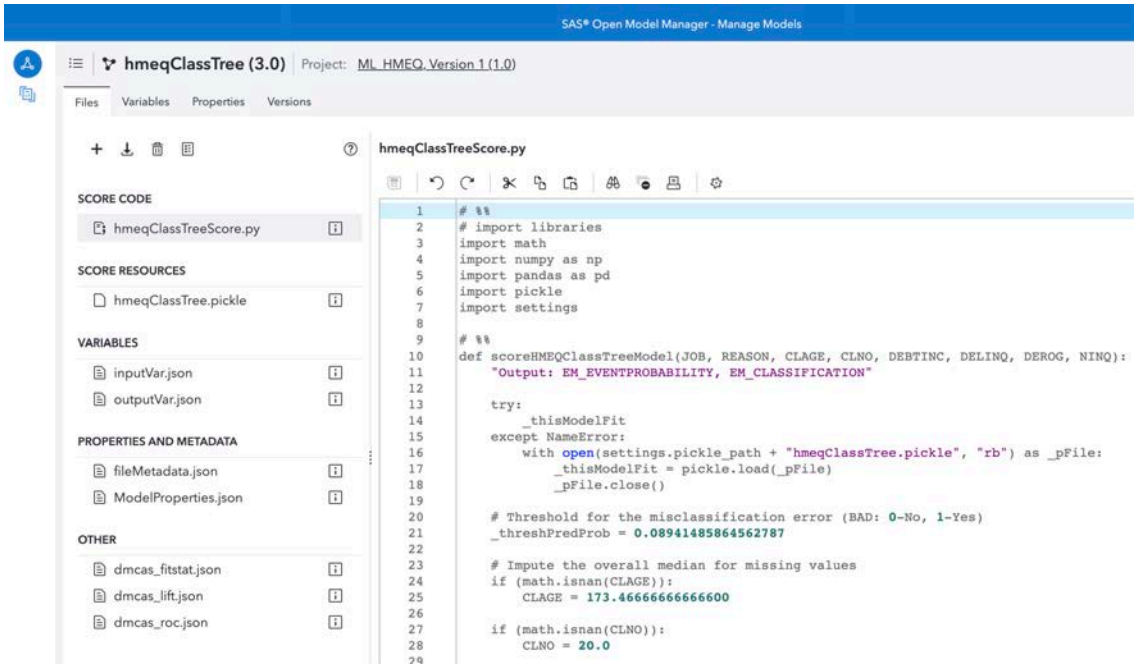


Figure 2: Imported hmeqClassTree model

TOOLING PYTHON CODE TO WORK WITH MODEL MANAGER

The goal of Model Manager is to make processing of open-source models seamless, for testing that means making it as easy to test a Python model as a SAS model. In the import example above, we notice the model consists of a score code file, hmeqClassTreeScore.py, a pickle file, hmeqClassTree.pickle, and other metadata files.

For executing Python models, Model Manager uses a SAS extension called PyMAS (https://go.documentation.sas.com/?docsetId=masag&docsetTarget=n0b478i3vsj1pqn1dht0ctsorlet.htm&docsetVersion=5.2&locale=en). PyMAS uses the SAS DS2 language to bridge the SAS and Python execution environments. In a full SAS Viya installation, PyMAS needs to be configured as specified in the documentation before it can be used with Model Manager. In Open Model Manager PyMAS is pre-configured.

With PyMAS properly configured, the model that was imported above is ready to be used for any Model Manager function, but it does include some specializations that a modeler would need to know about:

```
# %%
import math
import numpy as np
import pandas as pd
import pickle
import settings
# %%
def HMEQClassTreeScore(JOB, REASON, CLAGE, CLNO, DEBTINC, DELINQ, DEROG,
NINQ, YOJ):
    "Output: EM_EVENTPROBABILITY, EM_CLASSIFICATION"

    try:
        _thisModelFit
    except NameError:
        with open(settings.pickle_path + "hmeqClassTree.pickle", "rb") as
_pFile:
```

```
            _thisModelFit = pickle.load(_pFile)

    # Threshold for the misclassification error (BAD: 0-No, 1-Yes)
    _threshPredProb = 0.08941485864562787

    # Impute the overall median for missing values
    if (math.isnan(CLAGE)):
        CLAGE = 173.46666666666600
…
```

Notice that this code simply provides a single function *HMEQClassTreeScore*. This is the function that is called to get the score for a single input line. In the definition, the parameters of the score function match the column names of the input data set. When running the model, the variable names are matched with the columns of the input data set to assure that the input matches the function declaration. It is not necessary to pass all variables of the input data, only the subset that are used by the model.

The line immediately below the function declaration is a Python docstring that specifies the output variables:

```
"Output: EM_EVENTPROBABILITY, EM_CLASSIFICATION"
```

This specifies the names of the output variables that are returned by the scoring function. SAS infers the data type of the return values.

To keep the model flexible for internal execution of tests and performance monitoring, as well as for external execution in any published destination, Model Manager manages all score resources such as pickle files and libraries dynamically. To do this, the *settings* package was introduced. If the score code accesses a pickle file, package, or data file that is defined in the model, then the following steps should be followed. In the model, set the file type of pickle files to *Python pickle*, for any secondary code packages or included data files, set the file type *Score resource*.

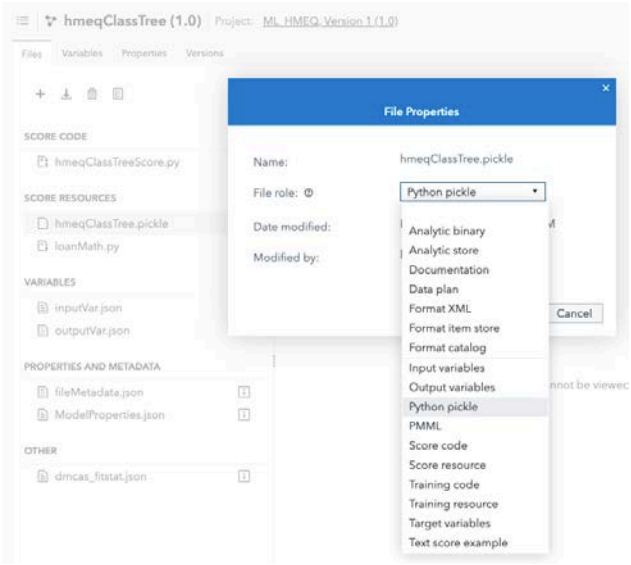


Figure 3: Setting a file type for a pickle file

In the above example, the model has the hmeqClassTree.pickle file and a secondary code package named loanMath.py. By setting the pickle to be of type *Python pickle* and the loanMath.py package to be of type *Score resource*, the score code can locate it by using this coding convention:

```
# above score method
import settings
import loanMath
…
# inside of score method
with open(settings.pickle_path + "hmeqClassTree.pickle", "rb") as _pFile:
```

The loanMath.py package only requires an import statement as it is placed in the main working directory for the model.

When necessary for code execution, Model Manager produces a code wrapper that is specific to the location where the code executes. These wrappers vary in code style. For example, when running a score test, Model Manager creates a code wrapper specific to the SAS DS2 Embedded Process score code type as seen in Figure 4. Other wrapper types are specific to DS2 Package code, and standalone container execution. Notice that the file *settings.py* is also generated.

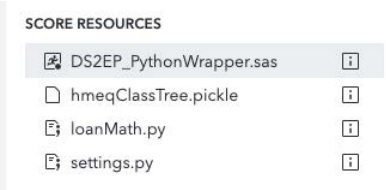


Figure 4: Score resources with DS2EP code wrapper

SELECTING THE BEST MODEL

With the hmeqClassTree model imported into Model Manager it is possible to compare it to other models. Part of the import process is to add the model into a project. In the code above the ML_HMEQ project was specified. That model already had two other models. With the hmeqClassTree model added, we are ready to evaluate which model should be used in the production system.

MODEL COMPARISON

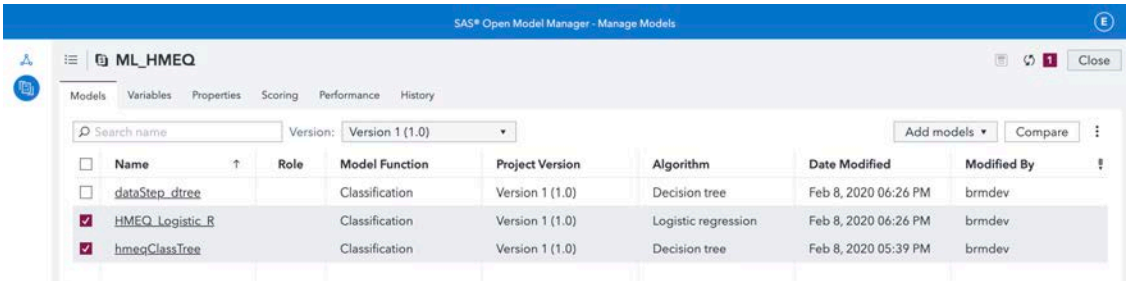
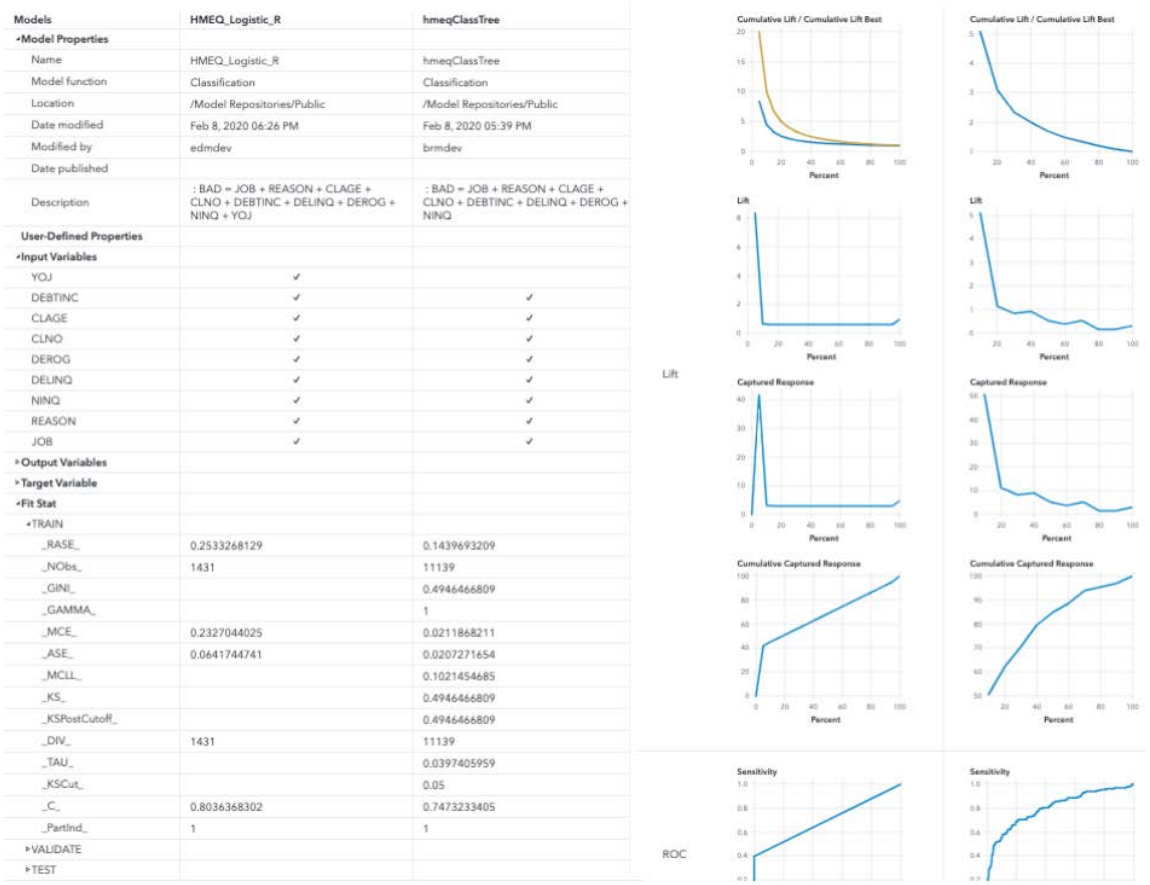


Figure 5: ML_HMEQ Project



Models	HMEQ_Logistic_R	hmeqClassTree
▾Model Properties		
Name	HMEQ_Logistic_R	hmeqClassTree
Model function	Classification	Classification
Location	/Model Repositories/Public	/Model Repositories/Public
Date modified	Feb 8, 2020 06:26 PM	Feb 8, 2020 05:39 PM
Modified by	edmdev	brmdev
Date published		
Description	: BAD ~ JOB + REASON + CLAGE + CLNO + DEBTINC + DELINQ + DEROG + NINQ + YOJ	: BAD ~ JOB + REASON + CLAGE + CLNO + DEBTINC + DELINQ + DEROG + NINQ
User-Defined Properties		
▾Input Variables		
YOJ	✓	
DEBTINC	✓	✓
CLAGE	✓	✓
CLNO	✓	✓
DEROG	✓	✓
DELINQ	✓	✓
NINQ	✓	✓
REASON	✓	✓
JOB	✓	✓
▸Output Variables		
▸Target Variable		
▾Fit Stat		
▾TRAIN		
RASE	0.2533268129	0.1439693209
NObs	1431	11139
GINI		0.4946466809
GAMMA		1
MCE	0.2327044025	0.0211868211
ASE	0.0641744741	0.0207271654
MCLL		0.1021454685
_KS_		0.4946466809
KSPostCutoff		0.4946466809
DIV	1431	11139
TAU		0.0397405959
KSCut		0.05
_C_	0.8036368302	0.7473233405
PartInd	1	1
▸VALIDATE		
▸TEST		

Figure 6: Model Comparison Results

In the ML_HMEQ project, we see three models including the one that we just imported. These models are all written in different languages but were trained using the same training dataset. In Model Manager the user can select multiple models and compare statistics associated with the training of the model.

In the example above, we are comparing our Python model with an R logistic regression model. We notice a slight difference in the input variables being used and see that each model provided a set of fit statistics, but the Python model provides more overall statistics. The charts show performance against the training data set. If validation and test data sets are included when generating the model, they are also included.

WORKING WITH CSV DATA

If you refer to the training example above, you notice that the training dataset is hmeq.csv. Models written in Python and R are typically trained and executed against data in CSV format. CSV data is easily handled by open-source data processing packages such as numpy and pandas. SAS, however, typically uses a proprietary data format. To help with the conversion there are several tools for integrating CSV data with Model Manager.

The easiest to use is the Data Explorer component in the Model Manager UI. This component is available as a standalone solution in the full SAS Viya installation. In Open Model Manager, the component displays when the user selects Manage data.

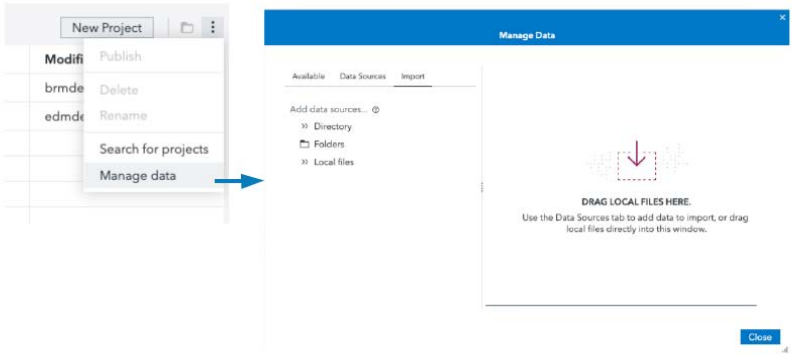
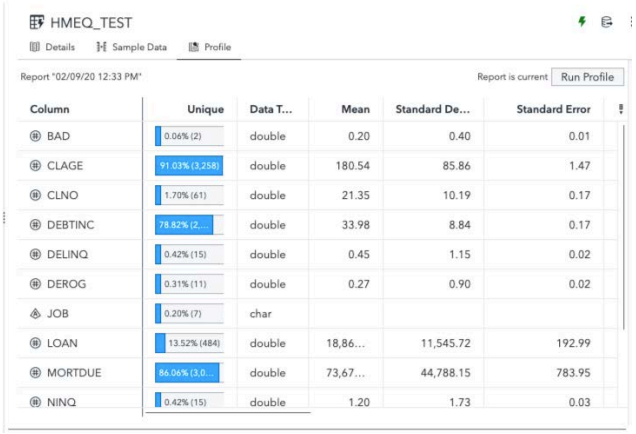


Figure 7: Managing Data

In the Manage Data dialog box, the user can import data sets, including CSV data, to be used with Model Manger's internal Testing and Performance features. The dialog box also enables users to get an overview of the data by viewing column details, sample data, or a data profile that includes useful statistics for each column such as the percentage of unique values, standard deviation, and standard error.



Column	Unique	Data T...	Mean	Standard De...	Standard Error
BAD	0.06% (2)	double	0.20	0.40	0.01
CLAGE	91.03% (3,258)	double	180.54	85.86	1.47
CLNO	1.70% (61)	double	21.35	10.19	0.17
DEBTINC	78.82% (2,...	double	33.98	8.84	0.17
DELINQ	0.42% (15)	double	0.45	1.15	0.02
DEROG	0.31% (11)	double	0.27	0.90	0.02
JOB	0.20% (7)	char			
LOAN	13.52% (484)	double	18,86...	11,545.72	192.99
MORTDUE	86.06% (3,0...	double	73,67...	44,788.15	783.95
NINQ	0.42% (15)	double	1.20	1.73	0.03

Figure 8: Data profile

The Data Explorer is a good way to make already cleansed and transformed CSV data available to Model Manager. But if your data is not fully prepared, SAS has Python packages to be included in your data preparation process to import the prepared data into the SAS data libraries. The saspy and swat packages are available on the SASSoftware GitHub site: https://github.com/sassoftware. Full documentation and examples for both are provided on GitHub.

TESTING THE MODEL

Once the candidate models have been selected, it is prudent to test the models and validate that the code does not produce errors when running against a testing dataset. Testing can uncover several issues, it can identify where the score code has syntax errors, or more commonly, where the score code does not respond well to variations in data or empty values.

Testing models in Model Manager is a simple process: select the model and the input dataset and request a test. As shown above, the appropriate code is generated to support executing the test. The testing output is similar for SAS models and open-source models.

Figure 9: Test execution results

In this example, the test failed. The log in the test results shows the following:

```
NOTE: Running DATA program on one node
ERROR: Traceback (most recent call last):
  File "/opt/sas/viya/home/SASFoundation/misc/embscoreeng/mas2py.py", line 944, in invoke
    out = up[1].get(func)[0](up[0])(*args)
  File "/tmp/tmpwwdizkys/model_exec_c4a0c99c-55b8-477f-9e91-7a5eae5916ca.py", line 8, in scoreHMEQClassTreeModel
    return hmeqClassTreeScore.scoreHMEQClassTreeModel(JOB, REASON, CLAGE, CLNO, DEBTINC, DELINQ, DEROG, NINQ)
  File "/models/resources/viya/brmdev_f01de479-ef91-464e-a409-cb695b19b4db/hmeqClassTreeScore.py", line 28, in scoreHM..
ERROR: Error reported by DS2 package pymas:
ERROR: DS2 "pymas" package encountered a failure in the 'execute' method.
```

Figure 10: Test result log

In the log we can see that there is a failure at line 28 of the score code. Looking at that code shows that it is attempting to replace missing values with an imputed value:

```
if (math.isnan(DEBINC)):
    DEBTINC = 34.81826181858690
```

While this appears to be a valid “not a number” check, it does not take into account when the value is passed as None. This is corrected by updating the code:

```
if DEBTINC is not None:
    if (math.isnan(DEBINC)):
        DEBTINC = 34.81826181858690
else:
    DEBTINC = 34.81826181858690
```

With this change for the DEBTINC column, the code executes successfully with our test data. Although this indicates that similar checks are likely needed for all of the numeric columns in our model. What this error shows, however, is a difference between how SAS DS2 internally translates empty values and how Python native packages such as pandas handle them. Running the first version of the code in a plain Python environment returns the correct output with no error. However, when executed through our score test, we get the error. Why?

The problem is with the internal data marshaling that is used for score testing. Since the score test uses a SAS DS2 wrapper, the data is marshaled by the DS2 data processor. In DS2, as opposed to pandas, empty values are treated as null instead of Nan. Because of this difference, it is best to perform a check for None as well as Nan when working with numeric columns. Similar processing is necessary for strings.

PUBLISHING MODELS TO A PRODUCTION ENVIRONMENT

In Model Manager, model deployment is termed “publishing”. Model Manager provides the capability to publish models to various environments. For Python models these include CAS, SAS Microanalytic Server (MAS), and Docker containers, which can be published to a private Docker registry or an Amazon Web Services account. For R models, only Docker containers can be published, but support for CAS and MAS is coming.

There are numerous ways that a published model can be added to a processing pipeline. This paper focuses on publishing to Docker containers. Deploying a model in a container is a relatively new advancement. Containers provide several advantages:

- Portability across systems
- Models are isolated from one another
- Each runtime environment can be customized to its model
- Transactions can scale to multiple containers

Model Manager publishes models to containers in the same manner that it publishes to other destinations. Administrators creator a container destination descriptor and when a user requests to publish a model, the container destinations are presented alongside the other destinations.

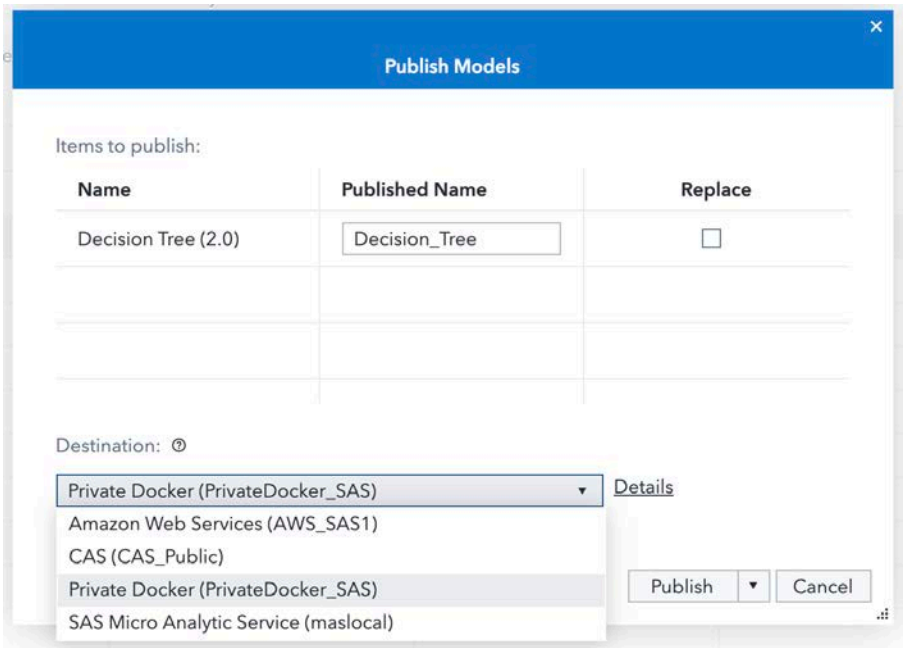


Figure 11: Model publishing to container destination

Once the model is published, the container can be verified from inside of Model Manager using the publish validation feature. When the publish completes successfully, a publish validation job is created that can be used to execute the container in the Docker runtime that was used for publishing.

Publish validation is performed in the same project panel for score testing.

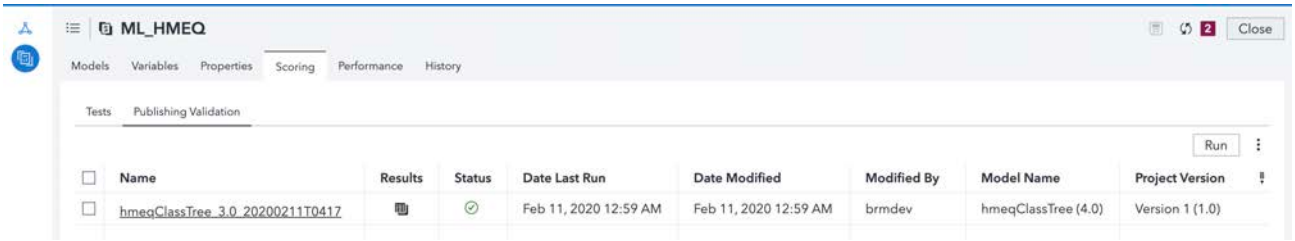


Figure 12: Publish verification results

For this example we can see that the sample data scored without any errors.

USING THE MODEL CONTAINER IN A DOCKER RUNTIME

While the test above shows that the container can be loaded and executed correctly, users do not execute the container only through Model Manager. For production usage, the container is used in an environment that is separate from Model Manager. Therefore, it is

necessary to understand how the container works in order to add it to a production environment.

As previously stated, the model container provides a standalone runtime for the model. The model containers published by Model Manager use a common web-service interface for interaction:

- GET http://container_url:8080/ -- returns “pong” to indicate that the container is available
- POST http://container_url:8080/executions -- executes the model for each line of data in the file
    - Body: file=csv data file
    - Result: JSON containing the test id and the http result status

      `{"id":"1581398995.1859481","status":201}`
- GET http://container_url:8080/query/<testId> -- retrieves the result csv
- GET http://container_url:8080/query/<testId>/log -- retrieves the execution log
- GET http://container_url:8080/system/log -- retrieves the system history

Once the container is published, it can be run in a Docker environment that is capable of running Debian linux containers. To use the container:

```
docker pull docker.repo.com/models/hmeqclasstree:latest
docker run -p 8080:8080 docker.repo.com/models/hmeqclasstree:latest
```

The above commands pull the container into your local docker storage and start the container. The web service calls are made on port 8080.

With the container running we can write a simple test program in Python to use it. This program is similar to the program that runs in the production pipeline. First, we need to prepare the sample data and store it to a CSV file.

Some CSV sample data for exercising the hmeqclasstree model:

```
"REASON","JOB","YOJ","DEROG","DELINQ","CLAGE","NINQ","CLNO","DEBTINC"
"HomeImp","Other", 7, 0, 2, 121.83333333, 0, 14, 0
```

Next we have a simple Python program that exercises the container:

```
import requests
import json
import sys

protocol = "http"
server = "server.mycompany.com"
   • port = "8080"

# Check that the container is available
pingResponse = requests.get(protocol + "://" + server + ":" + port + "/")
if pingResponse.text == "pong":
    print("Server running")
else:
    print("Server unavailable")
```

```
    sys.exit()

# Load the sample data and request a score
multipart_form_data = {
    'file': (open("HMEQ_Sample.csv", 'rb'))
}
response = requests.post(protocol + "://" + server + ":" + port +
"/executions", files=multipart_form_data)
print(response.json())
testId = response.json()['id']

# Get the result and log of the execution
response = requests.get(protocol + "://" + server + ":" + port + "/query/" +
testId)
print(response.text)

response = requests.get(protocol + "://" + server + ":" + port + "/query/" +
testId + "/log")
print(response.text)
```

And produces these results:

```
Server running
{'id': '1581404170.482587', 'status': 201}
BAD,LOAN,MORTDUE,VALUE,REASON,JOB,YOJ,DEROG,DELINQ,CLAGE,NINQ,CLNO,DEB
TINC,EM_EVENTPROBABILITY,EM_CLASSIFICATION
1,1300,70053,68400,HomeImp,Other,7,0,2,121.83333333,0,14,0,0.431818181
8181818,1

Scoring...
 python -W ignore ContainerWrapper.py -i /pybox/model/HMEQ_Sample.csv
-o 1581404170.482587.csv

Completed!
```

MONITORING THE PRODUCTION MODELS

As seen in the example in the previous section, when a model executes in a container it produces a CSV response with the desired response variables. This data is available to your production pipeline, where it is accumulated into a dataset and used by other processes in your business that are interested in the response. Since the accuracy of the responses decays over time, it is important to periodically monitor the responses against the real outcomes in the system.

Model Manager provides performance monitoring that can be added to the model / data pipeline. To use Model Manager's performance monitoring, users create a performance definition in the Model Manager UI. The definition has several parameters, such as the following:

- whether the model score columns are prepopulated or if the model should be executed when generating the performance report in order to populate the score output
- whether to use a single table of input data, or a set of tables where each is specific to a time span of usage
- the model or set of models that are pertinent to the report

For monitoring production models, it is best practice to use data that is already scored.

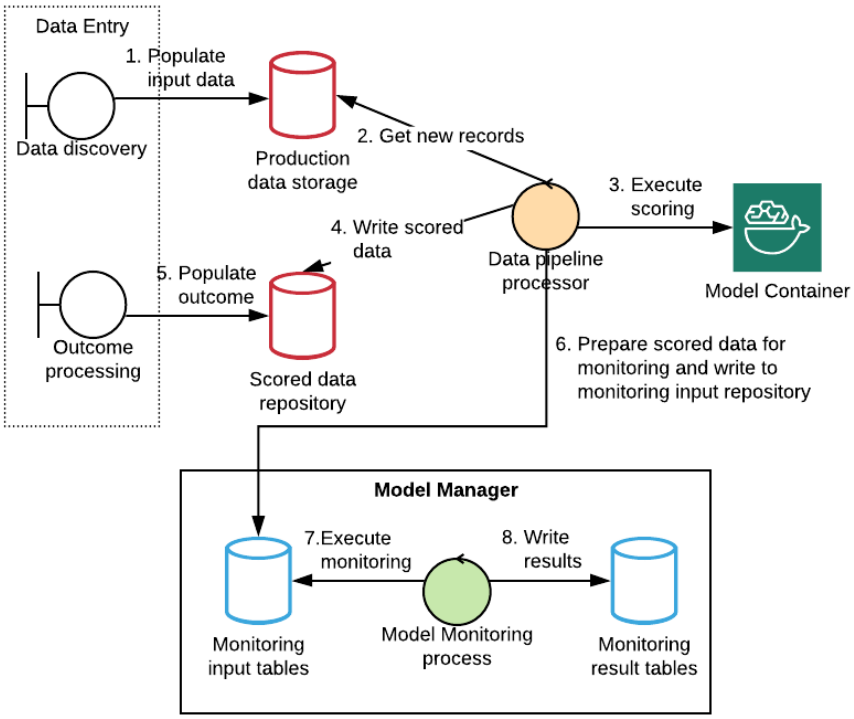


Figure 13: Example monitoring process pipeline

Figure 13 shows one possible processing flow. In this example, we define the monitoring process to use pre-scored data. Once in the scored data repository, an external process updates the data with the real outcome.

Once the real outcome is known, our pipeline processor prepares the scored data on a periodic basis and writes a separate table for each period to the monitoring input table store. The individual tables have a prefix identifier and then identify the associated period. For our hmeqclasstree model, we define a prefix of “hmeqtree”. Since our data pipeline processes new data quarterly it names the input data appropriately, for example, “hmeqtree_Q1”.

Figure 14: Performance monitoring report

Above is a sample performance report for four quarters of data. The performance report can be run manually in the Model Manager UI, it can also be scheduled using the Job Scheduler, or is accessible through the Model Manager API.

CONCLUSION

This paper addresses how to use Model Manager with open-source models as a core component of the model life cycle. With the techniques provided in this paper you can move a model from discovery to deployment and provide periodic monitoring.

Model Manager is continually evolving to provide support for more open-source model types and integration points. The deployment of models to Docker containers, cloud-enables the models and frees them from the standalone proprietary hardware of the past. This greatly broadens their scope but makes them much more difficult to manage and monitor. Model Manager simplifies this process.

REFERENCES

Xin, H., Jia, J., Duling, D., Toth, C. 2019. “Cows or Chickens: How You Can Make Your Models into Containers.” *Proceedings of the SAS Global Forum Conference.* Cary, NC: SAS Institute Inc. Available https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3489-2019.pdf

Hummer, W., et al. 2019. “ModelOps: Cloud-based Lifecycle Management for Reliable and Trusted AI.” IBM Research AI. Available http://hummer.io/docs/2019-ic2e-modelops.pdf.

CONTACT INFORMATION

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

Glenn Clingroth
SAS Institute Inc.
919-677-8000
Glenn.Clingroth@sas.com