

SAS[®] and Open-Source Model Management

Special Collection



Foreword by
Marinela Profi

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2020. *SAS® and Open-Source Model Management: Special Collection*. Cary, NC: SAS Institute Inc.

SAS® and Open-Source Model Management: Special Collection

Copyright © 2020, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-970170-81-8 (Paperback)

ISBN 978-1-951686-16-1 (Web PDF)

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

June 2020

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.

SAS software may be provided with certain third-party software, including but not limited to open-source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to <http://support.sas.com/thirdpartylicenses>.

Table of Contents

[Foreword](#)

[The Aftermath What Happens After You Deploy Your Models and Decisions](#)

By David Duling

[Turning the Crank: A Simulation of Optimizing Model Retraining](#)

By David Duling

[Open-Source Model Management with SAS® Model Manager](#)

By Glenn Clingroth, Hongjie Xin, and Scott Lindauer

[Deploying Models Using SAS® and Open Source](#)

By Jared Dean

[Cows or Chickens: How You Can Make Your Models into Containers](#)

By Hongjie Xin, Jacky Jia, David Duling, Chris Toth

[Choose Your Own Adventure: Manage Model Development via a Python IDE](#)

By Jon Walker

[Build Your ML Web Application Using SAS AutoML](#)

By Paata Ugrekhelidze

[Monitoring the Relevance of Predictors for a Model Over Time](#)

By Ming-Long Lam, Ph.D.

[Model Validation](#)

By Hans-Joachim Edert and Tamara Fischer

[Appendix](#)

Free SAS® e-Books: Special Collection

In this series, we have carefully curated a collection of papers that introduces and provides context to the various areas of analytics. Topics covered illustrate the power of SAS solutions that are available as tools for data analysis, highlighting a variety of commonly used techniques.



Discover more free SAS e-books!
support.sas.com/freesasebooks

 sas.com/books
for additional books and resources.

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. © 2017 SAS Institute Inc. All rights reserved. M1673525 US.0817


THE POWER TO KNOW.®

Foreword

When it comes to model development and deployment, organizations should have the freedom to choose different programming languages, tools, techniques, and run-time environments because only a modeling melting pot is what can fuel innovation and creativity.

Data Scientists develop models using different interfaces, algorithms, and tools. IT leaders adopt different environments and paradigms to execute analytics – on-premises, cloud, hybrid, through APIs, in real-time, in database, on server, on edge. It's what I like to call *analytical heterogeneity*, a status where analytics is not limited to one single methodology, tool, or algorithm but is able to leverage the full potential of the fast-growing and rapidly changing ecosystem of analytical solutions and technologies available, both open source and commercial.

However, a result of this differentiated ecosystem has been the increasing complexity of model life cycle management, and the difficulty of operationalizing models and start getting value from them. According to IDC, only 35% of organizations indicate that analytical models are fully deployed in production. Gartner states that over 60% of models developed with the intention of operationalizing them were never actually operationalized. Recent research by SAS indicates that 90% of models take more than three months to get into production. And 44% of models take over seven months. Too few models are getting into production, and for those models that do, it takes too long to influence decision making.

Without a structured and standardized process to integrate and coordinate all the different pieces of the model life cycle, analytical heterogeneity can turn into *analytical entropy*, a status where the usage of a diversified number of tools and technologies lacks governance, collaboration, traceability, oversight, monitoring and operationalization of models, thus resulting in chaos, cost increases and missed business opportunities.

Imagine what happens when the creator of a model moves into a new role or leaves the organization. How do you effectively maintain, monitor, and improve upon their work?

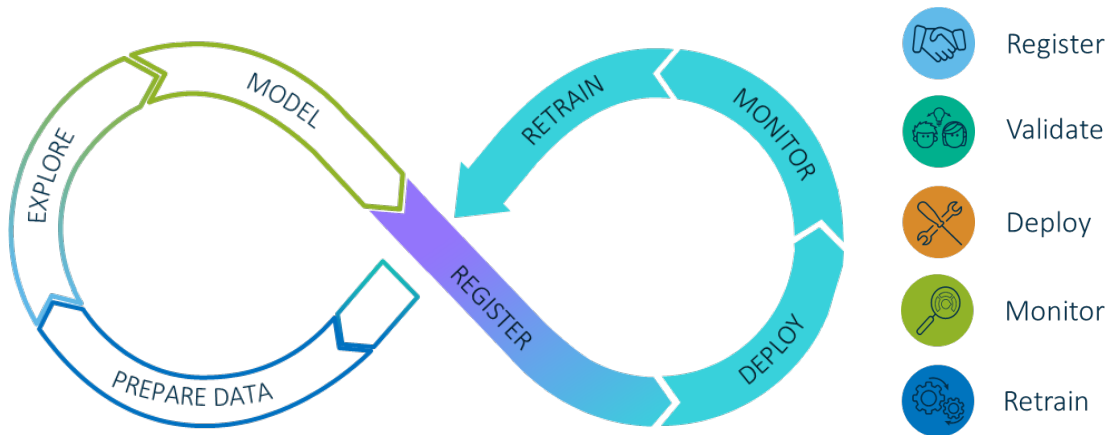
Imagine a company that has hundreds or even thousands of models for different business problems or use cases in different programming languages. How do you effectively manage versioning, reproducibility, deployment, scalability, testing, and governance?

The need to streamline and operationalize analytical models is the focus of the SAS Model Management solution. It enables organizations to adopt a model management strategy into their analytics life cycle that allows users to register, test, deploy, monitor, and retrain analytical models, uniting Data Scientists, IT/DevOps, and Business Analysts.

The goal is to provide analytical heterogeneity with the structure, standardization, coordination, and technology needed to turn it into a competitive advantage, thus enabling ongoing governance, traceability, transparency, and the ability to leverage any technology available.

Several groundbreaking papers have been written to demonstrate these techniques using SAS Model Management solutions. We have carefully selected a handful of these from recent SAS Global Forum papers to show how a modeling melting pot can become long-term business value using SAS. Enjoy!

SAS Model Management



[The Aftermath What Happens After You Deploy Your Models and Decisions](#)

By David Duling

We're making it easy to deploy your models and decisions to numerous run-time environments. However, the model life cycle doesn't end once the model is created. Rather, it is just the beginning of the important phases of model monitoring and analysis. This need extends to SAS models and open-source and Predictive Model Markup Language (PMML) models. In this demonstration, you learn techniques for analyzing model performance, integration with business metrics, and root cause analysis.

[Turning the Crank: A Simulation of Optimizing Model Retraining](#)

By David Duling

Model retraining is a common practice in the advanced model life cycle. However, the critical question is how do you know when you need to retrain the model? Once the model is retrained, how do we determine when we need to redeploy the model? Can we predict how long the model will be relevant? The answers can depend on one or more of many factors including calendar fluctuations, business cycles, data drift, model performance, expected benefit, and many others. Given those factors, we want to find the optimal points in time to retrain and redeploy a predictive model. This paper presents a simulation study of different strategies and techniques for optimizing model retraining with the goal of maintaining optimal business performance.

[Open-Source Model Management with SAS® Model Manager](#)

By Glenn Clingroth, Hongjie Xin, and Scott Lindauer

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.

[Deploying Models Using SAS® and Open Source](#)

By Jared Dean

In the excitement and hype around machine learning (ML) and artificial intelligence (AI) most of the time is spent in the model building. Much less energy is expended on how to take the insights from models and deploy them efficiently to create value and improve business outcomes. This paper will show a complete example using DevOps principals for building models and deploying them using SAS in conjunction with opens source projects including Docker, Flask, Jenkins, Jupyter, and Python. The reference application is a recommendation engine on a web property with a global user base. This use case forces us to confront security, latency, scalability, repeatability. The paper will outline the final solution but also include some of the problems encountered along the way that informed the final solution.

[Cows or Chickens: How You Can Make Your Models into Containers](#)

By Hongjie Xin, Jacky Jia, David Duling, and Chris Toth

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.

[Choose Your Own Adventure: Manage Model Development via a Python IDE](#)

By Jon Walker

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

[Build Your ML Web Application Using SAS AutoML](#)

By Paata Ugrekhelidze

Online loan applications are automated processes that allow people to quickly receive loans without unnecessary delays. Nowadays, automated decision-making is dominated by machine learning algorithms. However, algorithms require a lot of manual and technical capabilities to be practical and effective. SAS AutoML offers the ability to automate the development of machine learning algorithms. This article will demonstrate the implementation of AutoML in banking loan applications.

[Monitoring the Relevance of Predictors for a Model Over Time](#)

By Ming-Long Lam

This paper presents a novel approach to monitor model performance over time. Instead of monitoring accuracy of prediction or conformity of predictors' marginal distributions, this approach watches for changes in the joint distribution of the predictors. Mathematically, the model predicted outcome is a function of the predictors' values. Therefore, the predicted outcomes contain intricate information about the joint distribution of the predictors. This paper proposes a simple metric that is coined the Feature Contribution Index. Computing this index requires only the predicted target values and the predictors' observed values. Thus, we can assess the health of a model as soon as the scores are available and raise our readiness for preemptive actions long before the target values are eventually observed.

[Model Validation](#)

By Hans-Joachim Edert and Tamara Fischer

SAS model governance can be integrated into an analytical ecosystem that includes a diverse set of open-source tools and many different scripting languages next to SAS analytics. In total, there are four parts to this blog post series that describe this modeling scenario in detail. The [first post](#) describes some basic principles of the DevOps (or ModelOps) approach. The [second post](#) discusses the "test pyramid," which originally is an industry-standard for test design in software engineering. However, we think it's valuable in the area of analytical test design as well. The [third post](#) is more "hands-on" in nature. It describes how a model validation pipeline can be implemented in real life – using tools like Git, Jenkins – and SAS Model Manager of course. Finally, the [fourth post](#) addresses a purely analytical topic. It contains a detailed explanation of an algorithm called the feature contribution index (FCI), how it works and how it can be used with SAS.

We hope you enjoy this special collection and find valuable ideas to apply in your model management challenges. Please join the conversation with your peers by registering in the [SAS Developer Community](#) and [SAS Support Community](#), where you can ask questions, post answers and comments, and find the latest news, articles, APIs, and other resources on how to leverage the full potential of your favorite analytical tool using SAS. In addition, we encourage you to visit the [SAS GitHub](#) Resources page for the most popular developers repos, code examples, libraries and tools.

For more information and further reading, see the Appendix for recommended e-books, white papers, and additional assets that can help you in achieving a successful modeling melting pot.



Marinela is Global Marketing Manager for SAS Model Management solutions, prior to which she was a Customer Advisor for Advanced Analytics, supporting organizations across EMEA in achieving data-driven decisions.

Her background is a mix of Business Administration, Statistics and Marketing. She holds a BS in Economics and master's degrees in both Business Administration and Statistics. Marinela is Global Ambassador and Member of Women Tech Network, an organization that enables women's empowerment in tech through leadership development, professional growth, mentorship, and networking events for professionals.

Session 3496 - The Aftermath What Happens After You Deploy Your Models and Decisions

David R. Duling, SAS Institute Inc.

ABSTRACT

We're making it easy to deploy your models and decisions to numerous run-time environments. However, the model life cycle doesn't end once the model is created. Rather, it is just the beginning of the important phases of model monitoring and analysis. This need extends to SAS models and open-source and Predictive Model Markup Language (PMML) models. In this demonstration, you learn techniques for analyzing model performance, integration with business metrics, and root cause analysis.

INTRODUCTION

You can think of the lifetime of a model as having three major phases. In data preparation, the business operational data is transformed and loaded for targeted business analytics. In the discovery phase, advanced reporting, statistics, and machine learning models are developed for gaining insight into patterns and trends that influence the business. In the deployment phase, those models are used to make predictions in critical business processes that drive operational decisions. This is where the models provide their greatest benefit to the organization; however, it is also where most businesses encounter challenges in completing the analytics lifecycle.

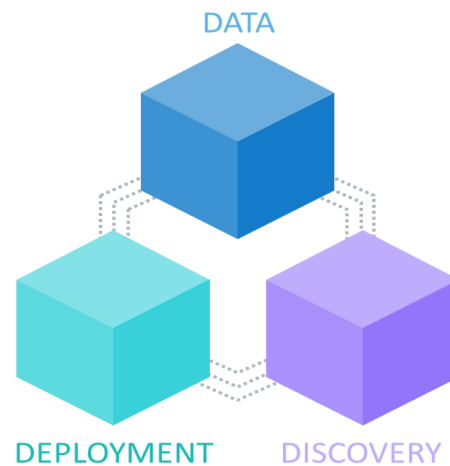


Figure 1. The new analytic life cycle.

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.

This paper will focus on model deployment and describe how models are used, how they are monitored, and how they can be evaluated and replaced.

MODEL SCORING

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 key component of most models is *score code* that can be evaluated with to make those predictions. Model scoring is the key function in model deployment. Score code is generally found in the primary language of the system that generated the score code, including the following forms:

SAS Procedures

This form was introduced with SAS® Stat® more than 30 years ago. While this is a very easy form to run in a SAS program, the drawback is that the code cannot easily be executed in non-SAS environments.

The original form of the output statement would replace missing values in the dependent column with new predicted values based on the model.

```
proc reg data= train outest=model;
    model bad= debtinc ninq clage clno;
    output out=scores;
run ; quit ;
```

Later, the Score procedure was introduced to score models that have a generalized linear model form. The model could be saved for use in scoring later without needing to re-create the model.

```
proc score data=train score=model out=scores type=parms;
    var debtinc ninq clage clno;
run;
```

Finally, the Score statement was introduced to some procedures such as proc Logistic to accept separate input and output data sets that were not included in the model training.

```
proc logistic data= train;
    class bad;
    model bad= debtinc ninq clage clno;
    score data= production out=scores;
run;
```

SAS Data Step

SAS® Enterprise Miner® introduced the concept of data step score code. SAS procedures were enhanced to include the code statement to produce score code. The following code shows creation of the SAS data step score code.

```
proc logistic data= train;
    class bad;
    model bad= debtinc ninq clage clno;
    code file= 'c:\temp\scorecode.sas';
```



```
run;
```

This form has some key advantages. The code is transparent and can be audited. Users can easily see the functional form and can reproduce the results in many different tests. The code can be modified and extended. Users can add additional logic for inline preprocessing data preparation such as missing value handling or binning, and for post processing such as model comparison or creation of decision variables. A portion of the data step code is shown below.

```
... snip ...
*** Effect: DEBTINC;
_LPO = _LPO + (-0.07795619150744) * DEBTINC;
*** Effect: NINQ;
_LPO = _LPO + (-0.11881501397556) * NINQ;
*** Effect: CLAGE;
_LPO = _LPO + (0.00596147711961) * CLAGE;
*** Effect: CLNO;
_LPO = _LPO + (-0.00263720328025) * CLNO;

*** Predicted values;
drop _MAXP _IY _P0 _P1;
_TEMP = 4.34723047453817 + _LPO;
if (_TEMP < 0) then do;
  _TEMP = exp(_TEMP);
  _P0 = _TEMP / (1 + _TEMP);
end;
... snip ...
```

SAS DS2

The DS2 language is a more structured form of SAS Data step with definitions for packages and methods that enables building more modular code. The primary scoring advantage is that DS2 language is portable and can be run outside of a SAS server in a variety of scoring environments. Generally, models are converted from Data step to DS2 for scoring outside the SAS server. SAS® Model Manager® will automatically convert Data step to DS2 as needed. SAS® VDMML Model Studio® is an exception that will create DS2 code for models that contain one or more Astore files.

SAS Astore file

Unfortunately, the size of data step files would be come too large with complex models that could produce well over ten thousand lines of data step code. The next solution was to package all the model structure information into a single binary file that could be used to later score new data. These files are named 'Astore' – Analytical Store format. You can execute an Astore model in a SAS program, a DS2 program, or directly in a CAS action. Astore files are portable and can be used in all scoring services produced by SAS. The following code shows both creation of the Astore file and usage of the Astore file.

```
proc gradboost data= casuser.train;
  target bad / level = nominal;
```

```

        input debtinc ninq clage clno / level=interval;
        savestate rstore=casuser.gradboost_model;
run;

proc astore;
    score data=casuser.train
        rstore=casuser.gradboost_model
        out=casuser.scores ;
quit;

```

PMML (Predictive Model Markup Language)

This format was developed by collaborative members of the Data Mining Group standards organization to facilitate the exchange of models between development software and run time scoring systems. A PMML file is not run time code, but rather a description of the model. You still need software to read, interpret, and then execute the model described by the PMML file. SAS® Model Manager® can read PMML files for most models that adhere to the PMML 4.2 standard. The file is converted to SAS Data step code that can be executed as part of a SAS program. In addition, SAS® Enterprise Miner® can create PMML files. The following portion of a PMML file shows the Logistic regression model coefficients for this sample. Refer to SAS documentation and DMG information for the range of models that can be expressed as PMML files.

```

... snip ...
<RegressionTable intercept="4.3472304745" targetCategory="0">
  <NumericPredictor name="DEBTINC" coefficient="-0.077956192"/>
  <NumericPredictor name="NINQ" coefficient="-0.118815014"/>
  <NumericPredictor name="CLAGE" coefficient="0.0059614771"/>
  <NumericPredictor name="CLNO" coefficient="-0.002637203"/>
</RegressionTable>
... snip ...

```

Python

Python has become the most popular open source language for building machine learning models. SAS enables use of Python in some SAS processes. Most Python models are saved as a Python Pickle file which is a general binary format for saving the state of a Python package. A Python program will read the Pickle file when scoring new data. The process is very similar to using an Astore file in a SAS program. The following program can run in SAS servers that support Python. Python execution is supported by the DS2 PYMAS package that is available with SAS® Model Manager®, SAS® Intelligent Decisioning®, and SAS® Event Stream Processing® (ESP). Python can also be executed directly by ESP. A Python program must be written in the following form with a function definition, the list of input variables, a comment that lists the output variables, and a return statement.

```

import pandas
import pickle
def scoreMNLogitModel (CLAGE, CLNO, DEBTINC, NINQ):
    "Output: I_BAD"
    # Open a read-only binary file for reading the pickle object
    _pFile = open('C:\\MyJob\\SGF\\2019\\MNLogit_Model1.pickle', 'rb')

```

```

    # Unpickle the file to recover the model specification and
estimates
    _thisModelFit = pickle.load(_pFile)
    # Close the binary file
    _pFile.close()
    # Construct the input array for scoring
    # the first term is for the Intercept
    input_array = pandas.DataFrame([[1.0, DEBTINC, NINQ, CLAGE, CLNO]],
        columns = ['const', 'DEBTINC', 'NINQ', 'CLAGE', 'CLNO'])
    # Calculate the predicted probabilities return data frame predProb
    _predProb = _thisModelFit.predict(input_array)
    # Determine the predicted target category
    I_BAD = pandas.to_numeric(_predProb.idxmax(axis = 1))
    return(I_BAD)

```

SCORING SERVICES

SAS provides several scoring services for interactive and operational systems. Table 1 shows a matrix of score code and scoring servers.

SAS

The traditional SAS® server can execute all types of score code. SAS can run interactively through Display Manager or SAS® Studio®, it can run batch jobs processing large tables of data, or it can run as a Stored Process for on-demand applications that don't require sub-second processing.

CAS

The SAS® Viya® CAS server is a big-data, in-memory server for complex data processing and analytical workloads. CAS can run all score code forms except for SAS procedures. CAS provides action sets for data step, DS2, and in-database processing.

INDB

SAS In Database processing is a feature of the SAS Access engines. For SAS® Viya®, In-Database processing is supported for Teradata and Hadoop, both Hive and Spark. This processing is normally used to process large tables of data that are natively created in the database. It is not generally appropriate for scoring transactions being processed by the database.

MAS

The Micro Analytic Score (MAS) service is a standard SAS® Viya® microservice that contains the ability to score models. MAS provides a REST interface that can be used to provide on-demand scoring functions that run in the sub-second time scale. The service is typically used in automation applications such as next best offer or fraud detection. MAS is distributed with the SAS® Model Manager® and SAS® Intelligent Decisioning® packages.

ESP

The SAS® Event Stream Processing® (ESP) server is a dedicated operational real-time system for processing high speed data streams. The server is typically used in IOT

systems that require embedded analytical processing. ESP can be programmed to detect patterns and execute actions, including scoring models.

Code Types	SAS Viya Scoring Services				
	SAS	CAS	INDB	MAS	ESP
Procedure	Y	N	N	N	N
Data Step	Y	Y	N	N	N
DS2	Y	Y	Y	Y	Y
PMML	Y	Y	Y	Y	Y
Astore	Y	Y	Y	Y	Y
Python	Y	Y	N	Y	Y

Table 1. Score code types and SAS® Viya® scoring services.

SENARIO

The remainder of this paper will refer to models created from data than is more complex than the simple examples shown above. The scenario is a fleet of trucks that are outfitted with sensors for collecting real time measurements of various systems on the truck.

Our task is to predict a future need for maintenance for each truck based on history of sensor readings and maintenance events. The target variable is maintenance_flag which has two values: 0 and 1. There are 12 fleets. Each fleet has from 1 to 8 trucks. The entire training data sample is 8307 rows of data. Each row of data contains a set of truck sensor measurements and a target variable that indicates if that truck later needed unscheduled maintenance. The data in Table 1 shows the distribution of the target variable for each fleet.

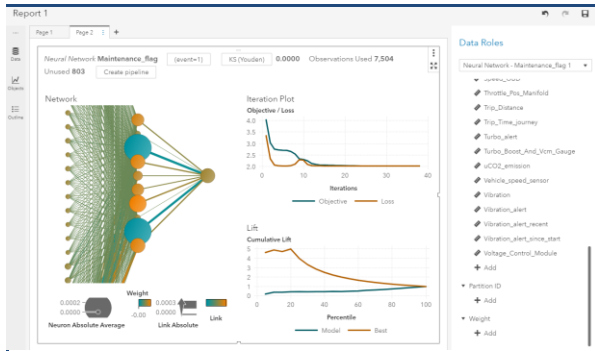
We can see that one fleet, 01013F1, did not have any truck maintenance events. However, we can build a model for the pool of all truck fleets and apply it to that fleet. This pattern demonstrates the power of using pooled data to improve models for all customers.

Table of Maintenance_flag by fleetid													
Maintenance_flag	fleetid												Total
	Fleet_00113F1	Fleet_00213F1	Fleet_00313F1	Fleet_00413F1	Fleet_00513F1	Fleet_00613F1	Fleet_00713F1	Fleet_00813F1	Fleet_00913F1	Fleet_01013F1	Fleet_01113F1	Fleet_01213F1	
0	479	305	300	1028	470	655	1044	378	674	23	442	624	6422
1	79	102	208	206	149	17	232	94	278	0	315	205	1885
Total	558	407	508	1234	619	672	1276	472	952	23	757	829	8307

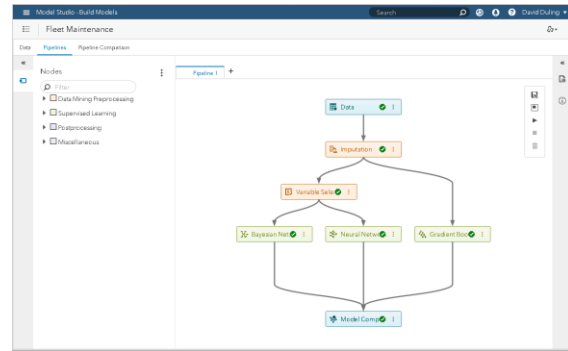
Table 2. Distribution of target values for each truck fleet in the data sample.

MODEL CREATION

While this paper does not primarily focus on model development, we do need a set of models to complete the scenario. We can start by exploring the data and building an initial model in SAS® Visual Statistics®. In this case we created a Neural Network model due to the flexible form. We then transferred the model to Model Studio where we have a more robust set of functions for building more models. We created a model pipeline to train and compare multiple candidate models. In addition to the Neural Network, we also created a Bayesian Network model and a Gradient Boosting Model. These models can be seen in displays 1 and 2. In addition, one of our developers is a Python enthusiast and he created a multinomial logistic model.



Display 1. Neural Network Model



Display 2. Model Studio pipeline

As the candidate models are trained, we can compare them in SAS® Model Manager®. We create one project for deploying this model. In that project, we can see all the candidate models including their attributes. We now have our set of models and can begin the process of model deployment.

Name	Role	Model Function	Project Version	Algorithm	Date Modified	Modified By
Bayesian Network (...)		Classification	Version 1 (1.0)	Bayesian network	Sep 16, 2018 11:51 AM	saschhd
Gradient Boosting (...)		Classification	Version 1 (1.0)	Gradient boosting	Sep 16, 2018 11:51 AM	saschhd
Neural Network (Pip...)		Classification	Version 1 (1.0)	Neural networks	Sep 16, 2018 12:00 PM	saschhd

Display 3. Model project showing multiple candidate models.

MODEL SELECTION

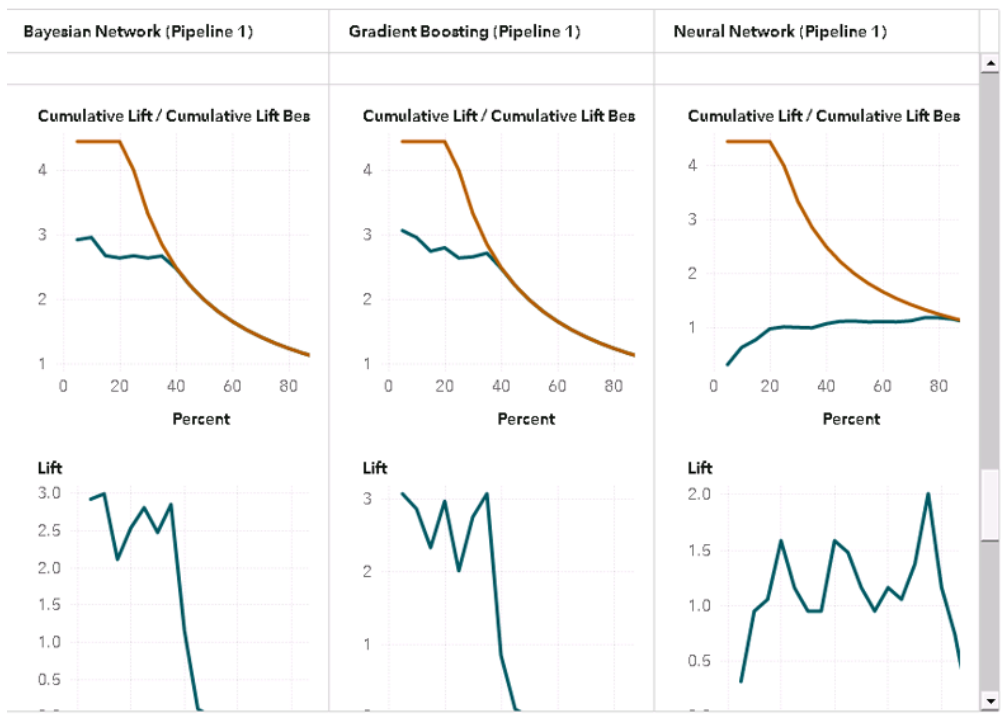
The first step in model deployment is to select and validate which model will be used in production. When the business process is being planned, there may be several candidate models that can be used. You will want the model the produces the best results, but you need to balance accuracy with other factors. Once you know your criteria, you can begin the process of model selection.

- Does the model match the business needs? If the model predicts individual sensor readings but you need the prediction of unscheduled maintenance, then you need to reconsider the model. It might a good enough proxy for the model you need, or maybe not.
- Does the model represent the correct time-period? If the model was trained on data from July but will be used in December, when the weather is very different, then you need to reconsider the model.
- How robust is the model? The model should be tested on data that represents the run time systems and model performance compared to both other models and the

original model training statistics. If the model performance significantly deviates, then you should reconsider the model.

- Can the model be explained? In many cases, the business requires the model to be explainable to external regulators, internal auditors, or customers. A great complex machine learning model might be useful for back-testing and measuring performance but might not be appropriate for production decision making.
- Does the model score code execute in the needed run time environments? Not all environments can run all score code types. The cost of recoding a model for a new language is very high. All the test processes must be repeated. Models with binary formats such as Astore files or Pickle files may not be portable to new languages.
- Is a new model a notable improvement over an older model? Each new model is a candidate model. It must be compared to the champion model on newer test data. If there is no improvement, then you need to reconsider the new model.
- Does the model require data that is available in the run time system? Some model features might not be available in all run time environments. In that case you need to know the cost to place that data in the run time system.
- How fast does the model execute? Some models might require more time to execute than others. This could be due to the functional form, or the quantity or availability of the needed data. If some of the data requires a fetch from an external database, that will cause a significant reduction in run time speed.

We can compare the attributes, variables, and statistics associated with each model. In this case, there is not a significant difference in the data variables needed for each model; however, there is a difference in the observed expected performance of the models. Display 4 shows the results. The neural network model shows signs of overfitting. The training data Cumulative Lift, shown in the brown plot line, is very good. The validation data plot, shown in the blue line, is much worse. It has an unusual convex shape indicating a potential missing data problem. This model is not robust and should not be used in production. The remaining models, Bayesian Network and Gradient Boosting, show similar performance in their training and validation statistics.



Display 4. Comparison of model statistics inside SAS® Model Manager®.

Another tool is model testing. We can score both models on a common test set and look at the results in detail for differences. To create the test set, we selected the first ten rows of data with target value 1 and next ten rows of data with target value 0. We need to make sure the models can predict these values correctly. In display 5, we have configured and executed the test for each model.

The screenshot shows the SAS Model Manager interface with the 'Fleet Maintenance' project selected. The 'Scoring' tab is active, and the 'Tests' sub-tab is selected. A table displays the results of two tests: 'Test_1' and 'Test_2'. Both tests were executed successfully, as indicated by the green checkmarks in the 'Status' column. The 'Test_1' results are for the Neural Network (Pipeline 1) model, and the 'Test_2' results are for the Bayesian Network (Pipeline 1) model. Both tests used the 'FLEET_TES T20' input table and were created on September 16, 2018.

Name	Results	Status	Model N...	Project V...	Input Table	Date Cre...	Date Co...	Created By
Test_1		⊙	Neural Network (Pipeline 1) (3.0)	Version 1 (1.0)	FLEET_TES T20	Sep 16, 2018 09:02 AM	Sep 16, 2018 09:02 AM	sasdhhd
Test_2		⊙	Bayesian Network (Pipeline 1) (3.0)	Version 1 (1.0)	FLEET_TES T20	Sep 16, 2018 10:04 AM	Sep 16, 2018 10:04 AM	sasdhhd

Display 5. Model Testing.

The results are clear as shown in Display 6. The Gradient Boosting model does not correctly predict the target value 1 in this sample. The Bayesian Network correctly predicts both target values and is the better choice for our deployment.

grad boost

bayes net

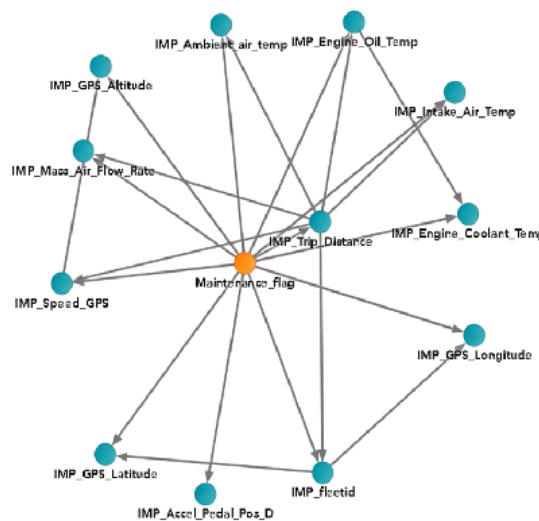
EM_CLASSIFICATION	EM_CLASSIFICATION
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0

Display 6. The results of model testing showing the better Bayesian Network.

Another benefit of the Bayesian Network in this case is that it is more explainable. This could be important in determining why the truck was referred for maintenance or looking for trends across the fleet. The information might be used in warranty claims. Display 7 shows the list of important variables for this model.

Order of Input Variables

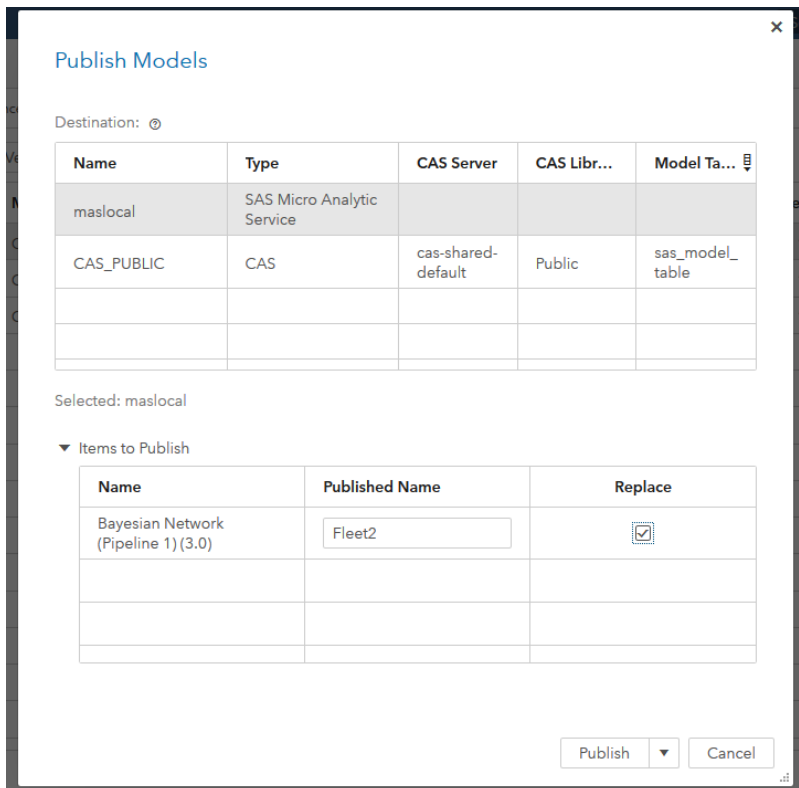
Variable Name	Order	Score
IMP_Voltage_Control_Module	1	-573.9700
IMP_Trip_Distance	2	-1,729.9008
IMP_Speed_GPS	3	-2,407.6189
IMP_GPS_Altitude	4	-2,480.7904
IMP_Ambient_air_temp	5	-2,492.9494
IMP_Intake_Air_Temp	6	-2,521.0292
IMP_Mass_Air_Flow_Rate	7	-2,530.6988
IMP_fleetid	8	-2,546.6659
IMP_Engine_Oil_Temp	9	-2,546.9421
IMP_GPS_Longitude	10	-2,548.6041
IMP_GPS_Latitude	11	-2,552.3625
IMP_Engine_Coolant_Temp	12	-2,590.1810
IMP_Accel_Pedal_Pos_D	13	-2,696.9639



Display 7. The important variables list and diagram for the Bayesian Network model.

MODEL DEPLOYMENT

Now that we have selected our model, we can deploy the score code it to our production server. In our configuration we have two choices. We can deploy the model to the CAS server for BI applications or batch processing. We can deploy the model the MAS service for online web service processing. In this scenario, we select the MAS service. As the trucks produce new batches of sensor data, new scores will be generated on demand and transmitted to the maintenance facility. When a high score is detected, the facility can schedule trucks for service visits.



Display 8. Model publishing to the MAS service.

The publish function will format the score code and transmit it to the MAS service where it will be compiled into memory and exposed in new web service REST endpoint. There is one more test we should run. We need to test the web service to make sure it produces the correct result. We can use the Publish Validation function in Model Manager to validate the endpoint. In this case, we are using the CAS table to supply the data, but the scoring function calls the MAS web service with the exact same interface that will be used by the business application. Use the same data used for model selection to ensure consistency. This test insures that the model is ready for use. Display 8 shows a set of tests that have already been run in both CAS and MAS to validate the production model. The results of the test will show that the correct scores have been returned by the service.

Name	Results	Status	Date Last Run	Date Modified	Modified By	Model Name	Project Version	Target Destinat...
Bayesian Network_Pipelin...		⊙	Sep 16, 2018 10:07 AM	Sep 16, 2018 10:06 AM	sasdhhd	Bayesian Network (Pipeline 1)(3.0)	Version 1 (1.0)	maslocal
Bayesian Network_Pipelin...		⊙	Sep 16, 2018 10:06 AM	Sep 16, 2018 10:06 AM	sasdhhd	Bayesian Network (Pipeline 1)(3.0)	Version 1 (1.0)	CAS_PUBLIC
Neural Network_Pipeline 1...		⊙	Sep 16, 2018 10:00 AM	Sep 16, 2018 10:00 AM	sasdhhd	Neural Network (Pipeline 1)(3.0)	Version 1 (1.0)	CAS_PUBLIC
Neural Network_Pipeline 1...		⊙	Sep 16, 2018 10:00 AM	Sep 16, 2018 10:00 AM	sasdhhd	Neural Network (Pipeline 1)(3.0)	Version 1 (1.0)	maslocal

Display 8. Testing the models before enabling the production process.

MODEL DEPLOYMENT

The main task now is to embed the model in the business application. The first thing we need to do is promote the model to the production fleet management server. SAS provides the Transfer Service to save and move content between servers. You may move all development artifacts including models and rule sets if you expect to make last minute changes in the production environment. If you expect to make no changes, then you can move only the MAS modules to the production server. This approach is less flexible but is more efficient and eliminates some chances of errors. Figure 2 illustrates the promotion of content between environments.

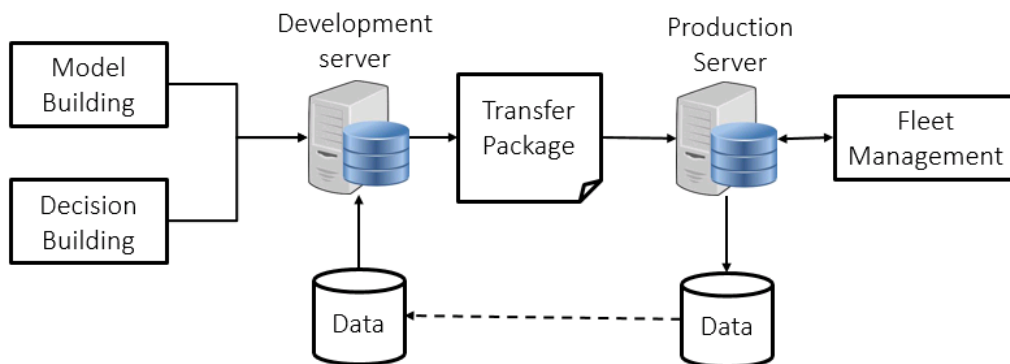


Figure 2. Promotion of models between environments.

In our scenario, we only need to move the MAS modules to the production server. The truck service management application will be programmed to call the MAS service to score new data observations and make optimal decisions. This could happen when the truck is stored overnight as daily data is downloaded and run through the scoring service. It could happen as the truck is moving and sensor measurement data is transmitted by LTE connection to the fleet data center. The fleet management application will receive the new,

run the model scoring service, and run sets of business rules to determine if action is needed and begin a workflow to manage the action. As a result, the truck might be scheduled for maintenance. This scenario is illustrated in Figure 3.

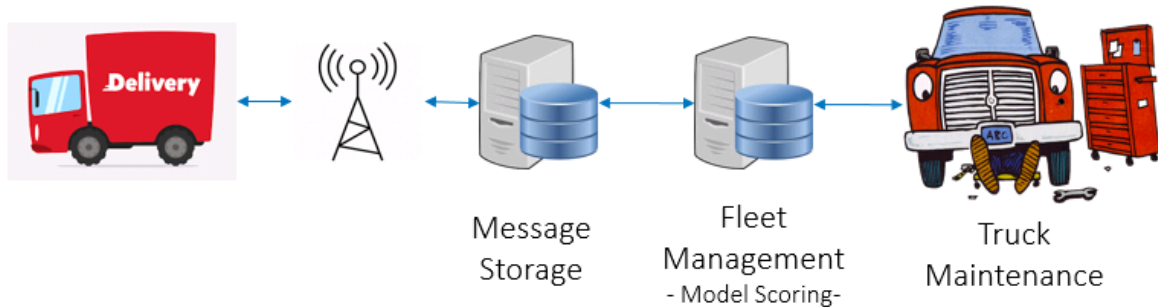


Figure 3. Fleet management system including model scoring.

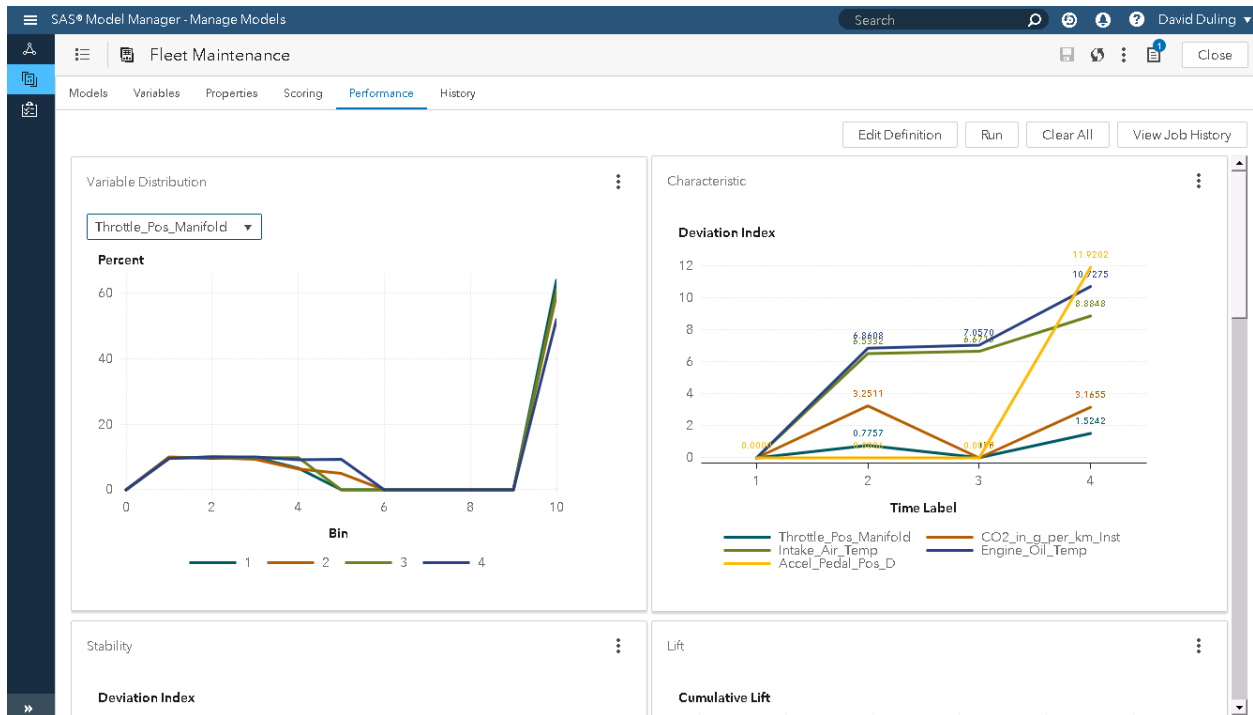
MODEL MONITORING

Once the model has been deployed in the production system, new data will be generated based on the model scores. This data should be stored in data sets for later analysis. This analysis is known as model monitoring. SAS® Model Manager® includes built in reports that compute the necessary measures for input and output data and the core fit statistics for classification and regression models. The first report we need to see is the variable distribution report which shows how the model input data is changing from the original model training data at subsequent each time-period. Variable change is important because small data value changes can have big effects on model accuracy. Each variable is divided into ten bins. The PSI is computed based on the deviation of proportions of data in each bin. We can name the original training sample **A** and the current scoring sample **B**.

$$\text{Deviation: } \Delta_i = 100 * (A_i / \mathbf{A} - B_i / \mathbf{B})$$

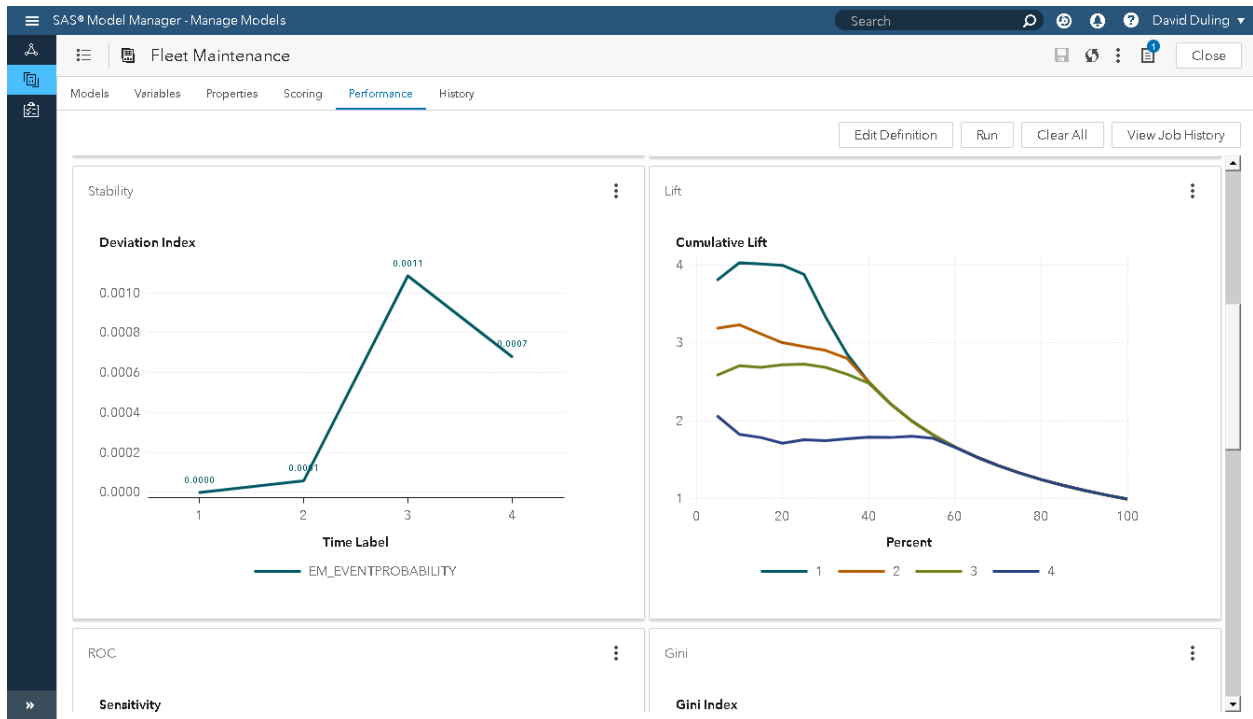
$$\text{Index: } \text{PSI} = \sum_i (\Delta_i * \ln(A_i / B_i))$$

In our truck fleet scenario, we are computing the model monitoring at four monthly time points. Each variable is divided into ten bins. In Display 9, the Variable Distribution plot shows the deviation in each bin for the Throttle_Pos_Manifold variable. In the Characteristic plot, the top 5 variables ranked by total PSI are shown at each time point (1,2,3,4). These variables are the ones most likely driving degradation in the model performance. The PSI is an absolute measure of deviation: larger values indicate greater amounts of deviation. The PSI does not show the increasing or decreasing direction of the deviation. One nice property is that PSI can be computed equally for continuous and categorical variables. You can use PSI to determine when the model is no longer reliable and needs to be replaced.



Display 9. Monitoring change in the distributions of predictor variables over time.

We can apply the same measures to the output of the models. Changes in the distributions of prediction or probability values also indicate that the data has changed indicating that the model might need to be replaced. We use the term Stability for deviation measures applied to model output. However, we can also measure model accuracy directly. After one month of time, the fleet management system will store records on actual truck maintenance events. At each event, we can determine if the truck needed maintenance for comparison to the most recent predictions by the models scores. We can measure model accuracy with model agnostic measures such as Lift, ROC, and Gini index. This calculation is the model accuracy portion of the model monitoring job. Display 10 shows both the stability measure of the predicted probability, and the Lift chart of model accuracy. Lift is a measure of relative model accuracy. The percentiles are created by ranking the model probabilities from high to low. In each percentile, the model accuracy is compared to the overall model accuracy. Higher values of lift indicate better predictive performance. In Display 10, we see both the model stability chart over the four time periods, and the model lift charts at each period. We can see there is a big change in stability in period 3 that is partially corrected in period 4. We can also see that model lift decreases in each period. These are clear indicators that the model performance is degrading.



Display 10. Model output value changes and accuracy changes over time.

MODEL ANALYSIS

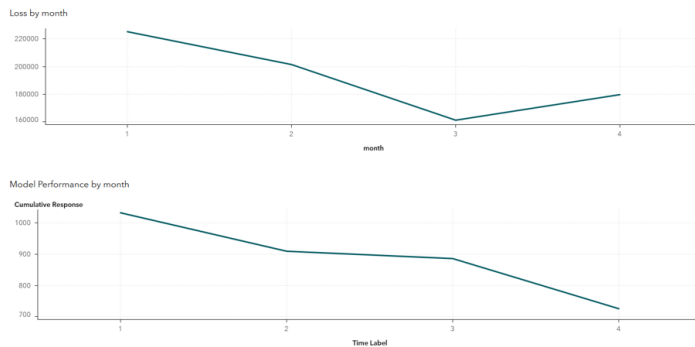
We can analyze change in model performance using standard statistical modeling tools that rank variable importance. This root cause analysis can reveal which factors are truly affecting change in accuracy. In the following analysis, we have created a new variable named Residual as the difference between predicted maintenance (0,1) and actual maintenance (0,1). We then model Residual based on the standard model predictors using a decision tree in SAS® Visual Analytics, as shown in Display 11. This model shows that the variables *Mass_Air_Flow_Rate*, *Engine_Oil_Temp*, and *Engine_Load* most contribute to model error. The business should ask their engineers to examine the factors influencing those variables to improve truck reliability.



Display 11. Root cause analysis of the model errors.

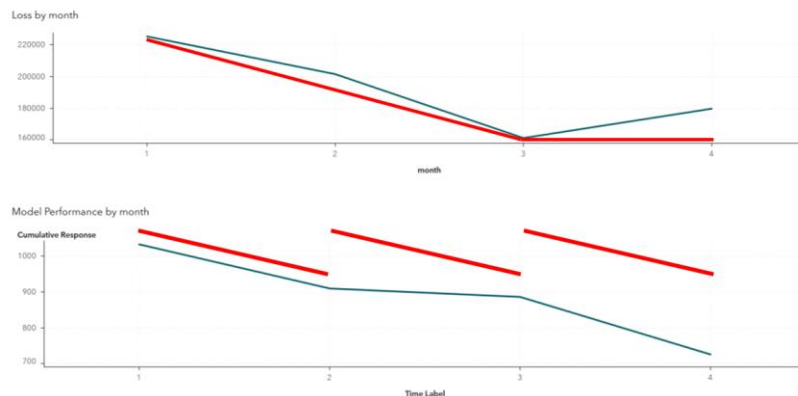
As a result, the business might also require the model be retrained to improve accuracy. The data scientist can trigger the retraining using SAS® Model Manager® for models developed in Model Studio or can directly reanalyze the data and produce new models. In either case, the models can be added to the original model as new versions. The model selection, testing, and deployment processes will then be repeated. The model monitoring jobs can then be executed in subsequent months and the performance of the project’s models can be plotted and analyzed over many generations of the deployed model. This view provides a long-term perspective on the impact of predictive models on the business outcome.

We can also directly visualize the impact of model performance on the business objectives. We have data of model performance over time generated by the model monitoring jobs. We can also accumulate data over time on business performance for comparison purposes. In this scenario, we have accolated data on the number of truck maintenance events, maintenance costs, and loss of income due to the loss of truck usage during maintenance events. In Display 10, we have plotted both model performance and business losses by month on the same scale. We can see that in months 1, 2, and 3, that business losses declined, but that model performance was also declining. This shows that we were still deriving benefit from the model despite the degradation. However, in month four, the business losses increased indicating that perhaps model inaccuracy is not contributing to predicted maintenance false positives resulting it excessive loss of truck usage.



Display 11. Comparison of model performance and business loss over time.

We can correct that chart by retraining the models. However, we should ask the question what can do to improve cumulative business performance. Each month of model accuracy data contains the information needed to retrain the candidate models. If we retrained each month, rather than waiting for more significant model error, the monthly degradation in model performance would not change, but the cumulative degradation model performance should be reduced. Figure 12 illustrates this ideal situation. The red line added to the chart shows the expected effect of retraining the model each month. At the start of each month, model accuracy has been restored to the level of model training. If no new data is introduced and no new and improved modeling method is introduced, each model retraining should achieve the same level of accuracy. At the end of the month model accuracy has degraded by the expected amount. In this case, cumulative business losses should decline to their minimal value and remain at that amount. Appropriately frequent retraining of the model should result in minimal business loss. There is a limit to this effect. You should not retrain a model when an insufficient amount of new data has been accumulated, which would result in a less accurate model. You should retrain the model at a rate that does not match the business cycle. For instance, if the trucks were running 24 hours a day in three shifts, then you might not want to apply a model trained on data from the overnight shift on measurements taken in the daytime when ambient temperatures are higher.



Display 12. Ideal model performance degradation and improved business loss.

CONCLUSION

The modern analytic life cycle of model creation, model deployment, and model monitoring provides a strong foundation for using machine learning and artificial intelligence to improve business performance. However, there are many details that need to be addressed to deploy models into automated business processes. Following a diligent process of model selection, testing, deployment, monitoring, and analysis can make the outcomes more reliable and efficient. This paper has demonstrated the SAS approach to controlling the model lifecycle, and choices the business can make to make the outcome more effective.

REFERENCES

- Lam, Ming-Long. 2019. "Monitoring the Relevance of Predictors for a Model Over Time." *Proceedings of the SAS Global Forum 2019*, Dallas, TX. Available at <http://support.sas.com/resources/papers/proceedings19/3448-2019.pdf>
- Chu, Robert, et al. 2009. "Dashboard Reports for Predictive Model Management." *Proceedings of the SAS Global Forum 2009*. Washington, DC. Available at <http://support.sas.com/resources/papers/proceedings09/045-2009.pdf>
- SAS Institute Inc. 2018. *SAS® Visual Analytics 8.3: Working with Report Data*. Cary, NC. Available at <https://go.documentation.sas.com/?cdcId=vacdc&cdcVersion=8.3&docsetId=vareportdata&docsetTarget=titlepage.htm&locale=en>
- SAS Institute Inc. 2018. *SAS® Model Manager 15.3: Performance Monitoring*. Cary, NC. Available at <https://go.documentation.sas.com/?cdcId=mdlmgrcdc&cdcVersion=15.2&docsetId=mdlmgrug&docsetTarget=n1t0lz86j0sd4vn11ta4jniiq5tk.htm&locale=en>
- SAS Institute Inc. 2018. *SAS® Visual Data Mining and Machine Learning 8.3: User's Guide*. Cary, NC. Available at <https://go.documentation.sas.com/?cdcId=vdmmlcdc&cdcVersion=8.3&docsetId=vdmmlug&docsetTarget=titlepage.htm&locale=en>
- SAS Institute Inc. 2018. *SAS® Micro Analytic Service 5.2: Programming and Administration Guide*. Cary, NC. Available at <https://go.documentation.sas.com/?docsetId=masag&docsetTarget=titlepage.htm&docsetVersion=5.2&locale=en>

ACKNOWLEDGMENTS

The author gratefully acknowledges the help of Ming-Long Lam at SAS Institute.

CONTACT INFORMATION

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

David Duling
SAS Institute Inc
David.Duling@sas.com
www.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.

Paper SAS4612-2020

Turning the Crank: A Simulation of Optimizing Model Retraining

David R. Duling, SAS Institute Inc.

ABSTRACT

Model retraining is a common practice in the advanced model life cycle. However, the critical question is how do you know when you need to retrain the model? Once the model is retrained, how do we determine when we need to redeploy the model? Can we predict how long the model will be relevant? The answers can depend on one or more of many factors including calendar fluctuations, business cycles, data drift, model performance, expected benefit, and many others. Given those factors, we want to find the optimal points in time to retrain and redeploy a predictive model. This paper presents a simulation study of different strategies and techniques for optimizing model retraining with the goal of maintaining optimal business performance.

INTRODUCTION

Most data mining studies focus on building the most accurate predictive models. Competition programs such as Kaggle often supply a single large data set and pose a unique prediction problem. The typical task is formed to create one predictive model with maximum test data accuracy. Competitive models are often formulas that have been carefully tuned to the unique objective function on the single large data set. Once the competition is completed, the supplier of the data harvests the knowledge created by the competitors. The competitors move on to the next challenge. However, data does not exist as a single point in time. In real-world applications, data is continuously collected from operational systems and is subject to changing conditions. The data collected in the second month may be different than the data collected in the first month. Therefore, we may need to create a new model in the second month or later. The process of creating a new model to adapt to changing patterns in the data is called **"model retraining"**. This paper expands on a sample of retraining strategies using a long running data sample from a publicly available source.

MODEL DECAY

In our 2019 paper "The Aftermath What Happens After You Deploy Your Models and Decisions", **we described how models are scored in an operational process**. We also concluded with a section on model decay and retraining, and then presented a theoretical example. Figure 1 shows two plots from that paper. In both plots, the lower green line shows a measure of model performance for a real model created on that data sample. The thicker red lines show theoretical forms of model decay. The top plot shows the ideal situation in the top red line showing a continuously high level of model performance versus the realistic situation with the descending line of model performance. This plot is unrealistic due to data drift and model decay, which refer to the natural process of making less accurate predictions due to changes in data over time. In the bottom plot, the top red line shows a more realistic situation where the model is retrained each month, restoring the predictive accuracy to the maximum expected level each month. The overall gain is much greater from the frequently trained models than the originally trained model shown in the lower green line in each plot. However, this is a theoretical example based on data that was over-sampled to create multiple time periods.

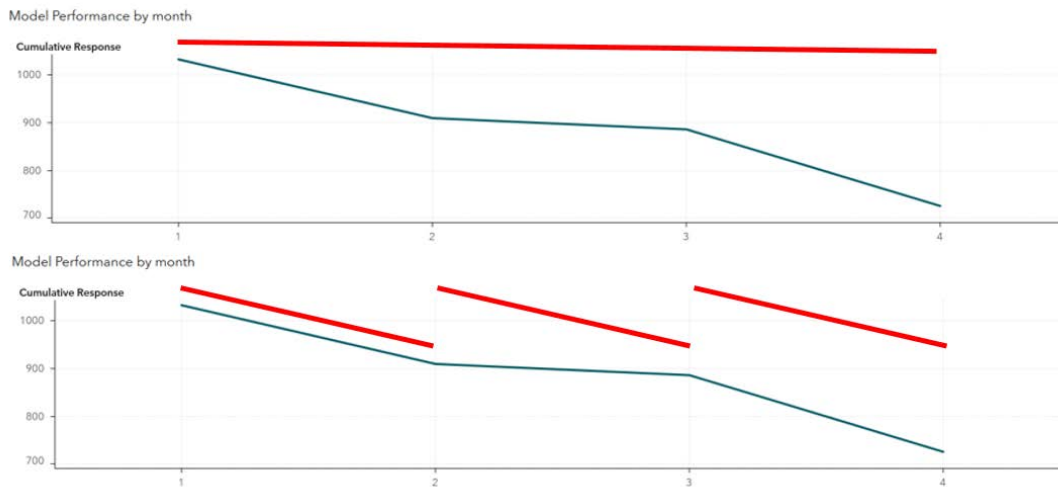


Figure 1. Theoretical Model Performance in Absence of Model Decay and by Frequent Retraining

MODEL MONITORING AND RETRAINING

There are numerous potential strategies for monitoring model performance and scheduling model retraining. Selection of a retraining strategy often depends on the business needs. Some processes can accept new models whenever they are created and validated. Some processes can accept new models only at fixed points in time. In many cases, models are created for comparison but never promoted to production. The main point is that model monitoring and retraining must be part of the business dynamic.

MONITORING

Model monitoring is a process for determining how well a model is or may be performing. There are several potential analyses that might be performed. Model monitoring process should measure all these factors:

- Data drift. Data values naturally changes over time due to numerous factors. People age. The economy becomes more or less positive. Mechanical parts erode or get updated. Competitors improve. Measuring changes in data values can be an early indicator of changes in model or business performance; however, not necessarily always.
- Model stability. Due to changes in data values, the distribution of model predictions may change. These changes will almost certainly impact business performance or planning. For instance, if predictions of truck maintenance-need increases, then more trucks will be scheduled for visits to the shop. More visits increase expenses regardless of the prediction accuracy.
- Model accuracy. If predictions target labels are available, then we may compute model accuracy measures. Degradation of model accuracy outside of acceptable bounds indicate a need for model retraining.
- Variable contribution. Changes in variable contribution to the model score or the model accuracy should be measured. These changes are also leading indicators of changes in model performance and may be used for reporting inferences about which variables caused changes in stability or accuracy. This may also be termed model interpretability.

The results of model monitoring should be stored and are used for model governance, statistical and business analysis, and as part of the process of determining if the model needs to be retrained.

RETRAINING

Model retraining is the process of recomputing a predictive or descriptive model on new data. Each new set of coefficients or effects is considered a new model. Models are retrained for multiple reasons.

- Business strategy. Changes to objectives such as increasing or decreasing acceptable levels of credit risk, investment in growth of new product lines, or numerous other facets will create the need for retraining models or creating new models.
- External conditions. Changes in business factors such as interest rates, new data sources, or suppliers of real-time truck metrics may create a need to retrain models.
- Business performance. Changes in measure such as response to promotions, credit repayment, truck repairs, and numerous others will create the need model retraining and / or review of the business strategy. Some change will be needed.
- Model Monitoring. Changes in the measures reported by model monitoring may create the need for retraining the model. This may be due to declining accuracy, data drift, or stability.

BUSINESS PROCESS

Organizations have many reasons for building predictive and descriptive models. Some models are used only for inference to learn more about the processes that shape the business or the expected impact of new strategies. Other models are created for integration into operational systems that interact with customer and business touchpoints to make the business more efficient, drive growth, improve loyalty, or other systematic objectives. The flow chart shown in Figure 2 is just one possible representation of a process for managing models.

The process flow is cyclical; however, we can say it starts with an operational business process that consumes and produces data. We are only representing the process for monitoring and retraining a model. We are not representing the process for defining a business problem and building the initial model. Here are the possible paths to start a model retraining process in this example:

- A timer event starts each cycle of the process, according to some predefined schedule.
- One timer event directly starts a model retraining. This is the process we are using in our simulation.
- Another timer event directly starts a model monitoring. This is the process we are using in our simulation.
- Another timer event checks for new data. If new data exists, a new monitoring job is executed. This could also be the process we are using in our simulation. New flight data arrives in monthly chunks.
- Regardless of the source, we always want the monitor process to record the current statistics for future analysis.

- A KPI measures computed from the monitoring output may drive the retraining. For instance, we may want to retrain if model accuracy falls below a threshold such as misclassification greater than 20%.
- A business strategy change may trigger a model retraining if not a completely new model.
- The newly retrained model should be tested for measures of robustness, accuracy, or expected ROI. It may be compared to a champion model. The model may fail testing and trigger a review of the model building process.
- If a new model passes testing, it may be deployed into the production environment for integration into the operational business process.

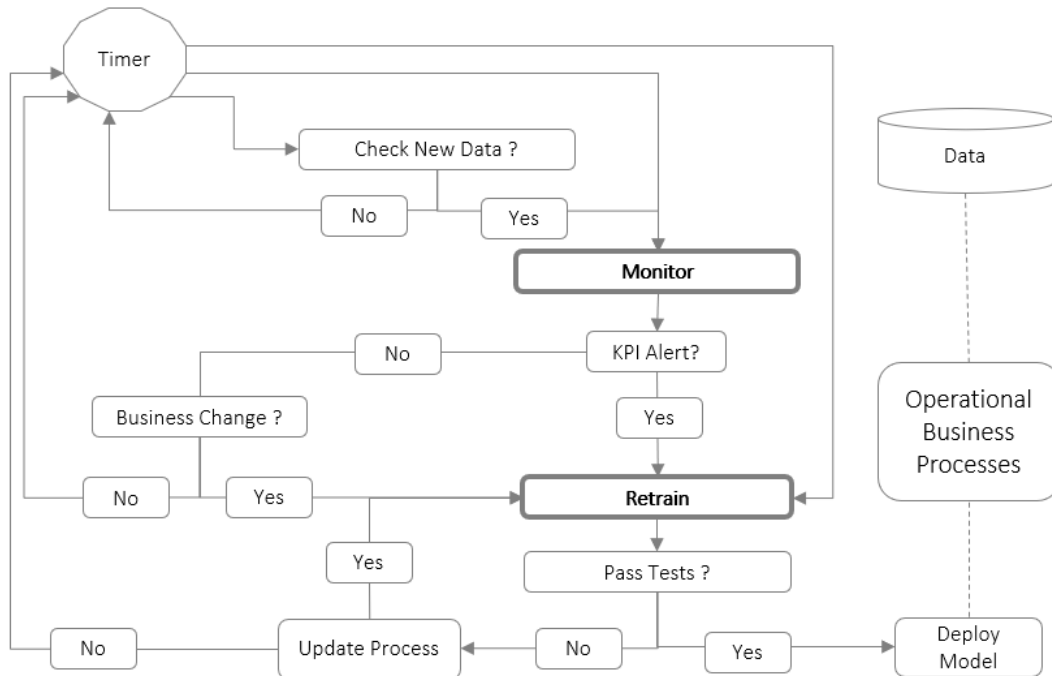


Figure 2. Sample Process Flow Diagram for Model Monitoring and Retraining

IMPLEMENTATION

The goal of our analysis is to test the effect of different strategies for model monitoring and retraining on long term model performance. To create this very custom process, completely new SAS code was written. Here are the descriptions for the major components of the code:

- The primary data was downloaded from the Bureau of Labor Statistics web site. The data consists of 145M rows of data stored in multiple CSV files.
- PROC IMPORT was used to import each CSV file into a corresponding SAS data set. Minimal data cleaning was performed at this stage. Several variables that are naturally numerical integers were mistakenly imported as character variables in the SAS tables and needed to be changed in the next step.
- DATA step and Base SAS procedures were used to transform and clean the data. A small number of observations had missing values for departure or arrival time and

were removed from the data. Several character variables were transformed into numerical columns. Variables that were irrelevant to the analysis were dropped. Variables that were proxies for flight late arrival were dropped. The target variable *LATE* was created with numerical Boolean value of (*Arrival_Delay* > 15). All months of data were **combined into one large table with 145M observations**. This “big table” was for all calculations.

- DATA step was used to create Training and Monitoring samples by querying the big table for specific months of data. The Training data was divided randomly into approximately equal samples of Train and Test data. Train data was used to build the model. Test data was used to report the statistics from the training exercise.
- SAS High-Performance Analytics procedures were used to create Decision Tree and Logistic Regression models. Default settings were used in all cases. Score code was saved from each training run into a directory of files. The score code was used to compute test data statistics and for model monitoring. PROC HPSPLIT and ODS were used to create the Decision Tree display images.
- Base SAS procedures were used to test statistics and model monitoring statistics such as mean monthly values of *Late* proportion, Probability, Misclassification, and True Positive rates.
- PROC SGPLOT and PROC PRINT were used to make all graphs and table displays.
- The SAS macro %SIM was used to script these operations. The %SIM macro was developed to simulate model retraining and monitoring with different time periods for the entire 303 months of data. All statistics used in this paper came from the %SIM macro.

Note: All SAS code that was used for this paper is available from the author upon request.

DATA

For the remainder of this paper, we will refer to the Airline flight data used in several data mining competitions and samples. The data is freely available from the U.S. Bureau of Transportation Statistics. The data starts in October of 1987 and continues to be updated. Our sample ranges from 1987 until the end of 2012. The data contains variables describing various attributes about the flight including the scheduled arrival time and the actual arrival time. Several papers have been written about this data including a visualization paper by Rick Wicklen as contribution to the ASA Data Expo contest in 2009.

We use this data because it represents a consistent source of data over many years, which has the potential to show change in data values and patterns over that time. In this exercise, our goal is to show long term trends in model monitoring; we are not trying to infer new knowledge about the data or build the most sophisticated model. Our sample contains 145,664,836 observations. All variables that would not be available at the time of model building or model deployment have been rejected. The first ten rows of data are printed in Table 1.

Obs	late	ORIGIN	DEST	UNIQUE_CARRIER	DAY_OF_WEEK	DAY_OF_MONTH	MONTH	YEAR	DISTANCE	CRS_DEP_TIME	CRS_ARR_TIME	CRS_ELAPSED_TIME	DEP_DELAY
1	0	JFK	LAX	AA	4	1	10	1987	2475	900	1152	352	1
2	0	LAX	HNL	AA	4	1	10	1987	2556	1300	1535	335	4
3	0	LAX	JFK	AA	4	1	10	1987	2475	830	1640	310	8
4	0	OGG	HNL	AA	4	1	10	1987	100	2035	2110	35	3
5	0	JFK	LAX	AA	4	1	10	1987	2475	1200	1446	346	2
6	0	DFW	HNL	AA	4	1	10	1987	3785	945	1250	485	2
7	0	HNL	OGG	AA	4	1	10	1987	100	1345	1430	45	-2
8	1	IAH	DFW	AA	4	1	10	1987	224	737	843	66	0
9	0	DFW	STL	AA	4	1	10	1987	550	825	1010	105	1
10	0	HNL	DFW	AA	4	1	10	1987	3785	1929	735	426	0

Table 1. Sample of Data Showing Variables with Typical Values

The derived target variable is named *late* and is either 0 or 1 to indicate more than 15 minutes late. The variables starting with CRS are scheduled times. The only variable that depends on the instance of the flight is departure delay, *DEP_DELAY*, which is necessary to produce good models without creating complicated lag variables.

The business of managing flight on-time performance has many latent factors. Airlines are reported to implement procedures to control and improve their on-time percentage as needed. They may use this data to make announcements about their performance and enhance their marketing campaigns. Flights that leave late may spend more fuel in an effort to regain time. Flight crew and airport expenses may constrain on time performance.

Table 2 shows the number of flights aggregated by month over the entire time period. Column N refers to the total number of flights. The monthly late rate averages 19.0% and ranges from 10.2% to 32.0%. Numeric model input predictor variables are also shown. The scheduled elapsed time, *CRS_ELAPSED_TIME*, shows a notably small standard deviation, perhaps indicating there has been little overall change in the scheduled routes.

Variable	N	Mean	Std Dev	Minimum	Maximum
late	145664836	0.1904100	0.0443149	0.1029323	0.3204210
DISTANCE	145664836	710.7584869	46.5993466	587.8776336	788.1398129
CRS_DEP_TIME	145664836	1332.88	26.8047265	1173.91	1361.85
CRS_ARR_TIME	145664836	1492.18	30.0217495	1303.90	1518.86
CRS_ELAPSED_TIME	145664836	122.9232413	8.2599934	99.5262048	136.6999134
DEP_DELAY	145664836	8.1111819	2.9086702	2.2205164	17.1477836

Table 2. Aggregated Monthly Means for the Entire Period of 303 Months

The plot of the number of flights per month is more interesting, in Figure 3. The sample contains 303 months of data over 25 years. The small yearly seasonality is apparent. There are peaks in travel around the winter holidays and over the northern hemisphere summer vacation periods. Markers have been added for selected significant global events. The dramatic impact that the September 11, 2001 terror attacks had an obvious impact on air travel, as expected, followed by a dramatic rise in the number of flights in January 2003. The rate of flights that are late each month is shown in the lower plot. There is minimal correlation between the total number of flights and the rate of late flights.

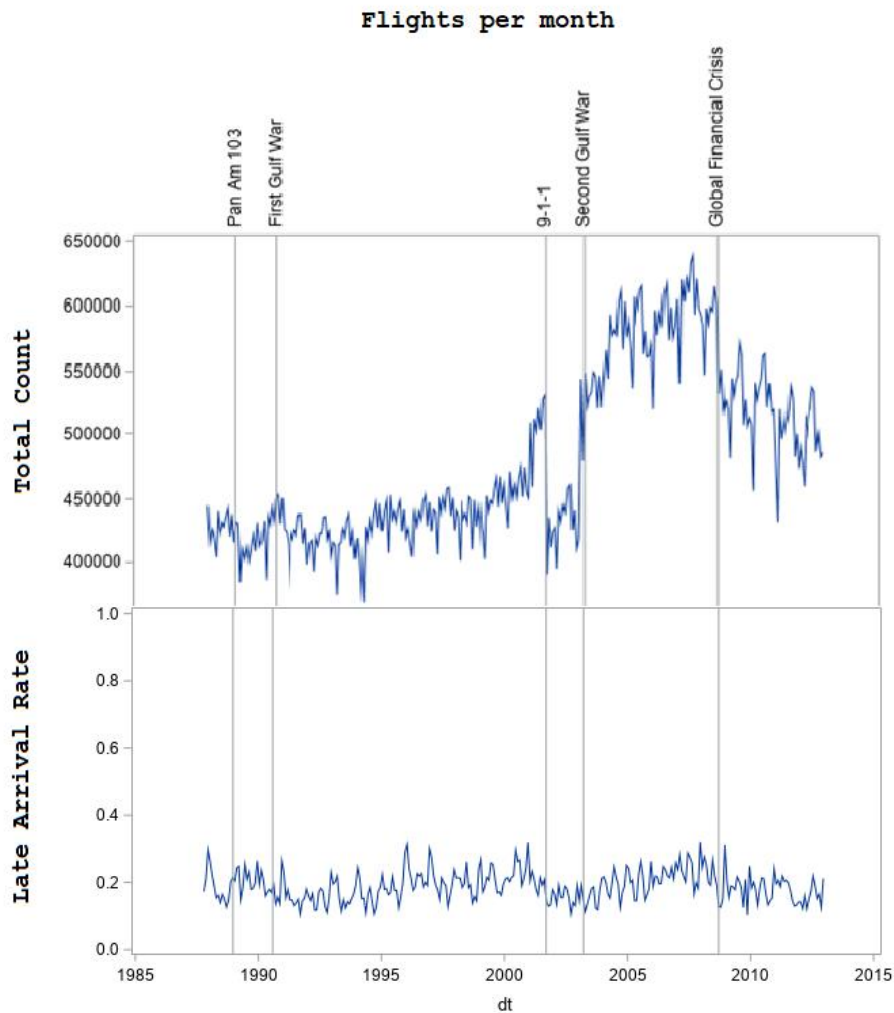


Figure 3. Total Number of Airline Flights per Month with Significant Event Markers

Most treatments of this data focus on modeling or visualizing the entire data set. However, imagine that you are an analyst working in 1987.

MODELS

The data is provided in monthly data sets. We created our first model on the first month of data, October of 1987, which contains 448620 rows. The data is randomly split into half training data and half test data. The model is a default decision tree created by PROC HPSPPLIT, which uses 10-fold cross validation to control the growth of the tree. A decision tree is good default model for this study since it is tolerant to new data values and naturally incorporates variable selection. Figure 4 illustrates the model results with the complete classification tree for the first month of data and the top subtree with details about the variables used in the model.

The complete classification tree demonstrates a complex model using several variables. The categorical variables identifying the airline, origination airport, and destination airport have higher cardinality and contribute to many of the tree branches. The subtree view shows top portion of the tree where departure delay is the most significant variable, as expected, but that other variables contribute to the classification values.

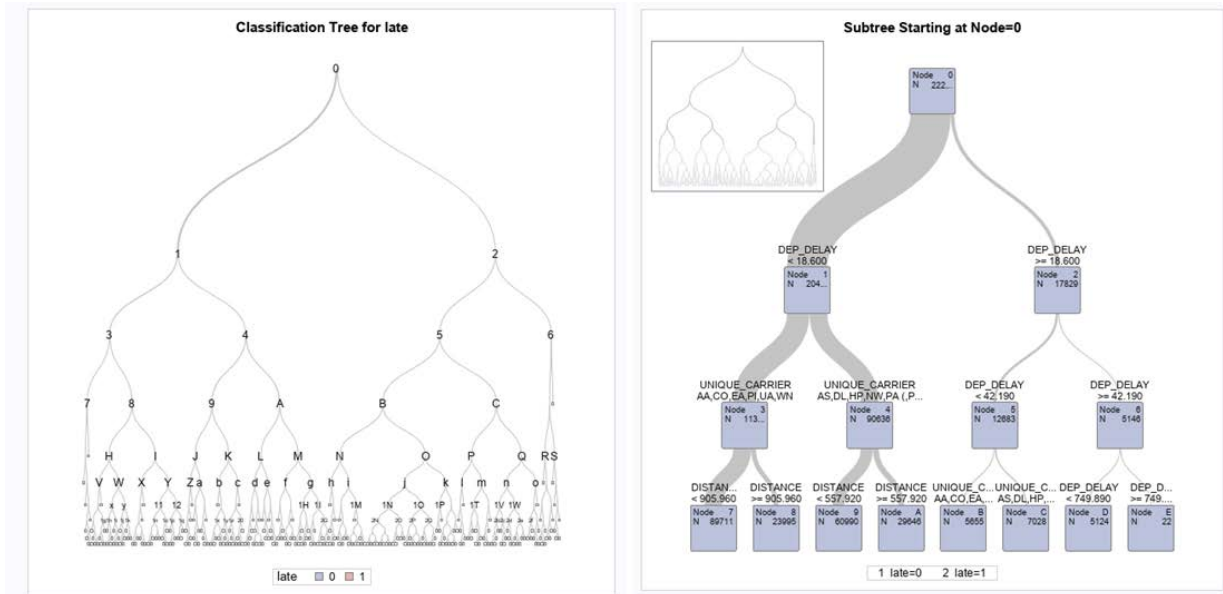


Figure 4. Classification Tree and Subtree of Model Variables

The relative variable importance values are shown in Table 3. These values correlate with the detail view of the decision tree. After *DEP_DELAY*, the remaining variables retain significant impact on the classification rates.

Variable Importance			
Variable	Training		Count
	Relative	Importance	
<i>DEP_DELAY</i>	1.0000	143.6	4
<i>UNIQUE_CARRIER</i>	0.1955	28.0687	18
<i>DAY_OF_MONTH</i>	0.1588	22.8046	71
<i>DISTANCE</i>	0.1517	21.7843	20
<i>CRS_ELAPSED_TIME</i>	0.1040	14.9300	28
<i>ORIGIN</i>	0.0916	13.1512	24
<i>DEST</i>	0.0907	13.0277	37
<i>CRS_DEP_TIME</i>	0.0898	12.8917	29
<i>CRS_ARR_TIME</i>	0.0831	11.9399	20
<i>DAY_OF_WEEK</i>	0.0285	4.0969	6

Table 3. Decision Tree Variable Importance Measures

Table 4 demonstrates the Decision Tree model results on the test sample from the first month of data. Late is the proportion of late flights, *i_late* is the proportion of flights classified as late, *i_misc* is the overall misclassification rate, and *i_tp* is the proportion of flights correctly classified as late. The score code was then applied to the test data sample and the classification (*i_late*), misclassification flag (*i_misc*), and true positive flag (*i_tp*) were computed. PROC MEANS was run to summarize the test scores and produced the data shown in Table 4. We can see that in the test sample, 17.3% of the flights were late, 7.2%

of flights were misclassified, and 10% of flights were correctly classified as late. These metrics will be used to monitor model accuracy in the remaining data.

First month model
The MEANS Procedure

Variable	Label	N	Mean	Std Dev	Minimum	Maximum
late		222609	0.1727199	0.3780058	0	1.0000000
P_late0	Predicted: late=0	222609	0.8273965	0.2284448	0	1.0000000
P_late1	Predicted: late=1	222609	0.1726035	0.2284448	0	1.0000000
i_late		222609	0.0809446	0.2727507	0	1.0000000
i_misc		222609	0.0721220	0.2586903	0	1.0000000
i_tp		222609	0.1005979	0.3007962	0	1.0000000

Table 4. Decision Tree Model Results on Test Sample from the First Month of Data

For comparison purposes, we also ran a Logistic Regression model through the same process. The absolute results are similar as shown in table 7. The misclassification rate is 1.2% higher, and the true positive detection is 1.5% lower.

First month model -- Logistic Regression
The MEANS Procedure

Variable	Label	N	Mean	Std Dev	Minimum	Maximum
late		222609	0.1727199	0.3780058	0	1.0000000
P_late0	Predicted: late=0	222609	0.8277114	0.2414860	6.11583E-102	1.0000000
P_late1	Predicted: late=1	222609	0.1722886	0.2414860	0	1.0000000
i_late		222609	0.1029249	0.3038614	0	1.0000000
i_misc		222609	0.0878311	0.2830498	0	1.0000000
i_tp		222609	0.0848888	0.2787167	0	1.0000000

Table 5. Logistic Regression Model Results on Test Sample on the First Month of Data

The results are not as good as the Decision Tree. Table 6 was generated to compare the two models. The table illustrates the comparison of models on first month on test data. Variable *ms* is the sequential month counter. TPR is the sample true positive rate. Decision Tree is champion based on misclassification and true positive rates. We can conclude that we have a valid modeling process using the default decision tree and will use that for the following results.

First month models - comparison

model	_FREQ_	ms	late	i_late	i_misc	i_tp	TPR
Decision Tree	222609	1	0.17272	0.08089	0.072279	0.10044	0.58152
Logistic Regression	222609	1	0.17272	0.10292	0.087831	0.08489	0.49148

Table 6. Comparison of Models on First Month on Test Data

It would have been tempting to build models on the entire data that account for all the seasonality, long term trends, forecasts, and significant events ahead of time. However, we must put ourselves into the position of the analyst in November of 1987 who received this minimal data with the task of producing models that would give the best prediction for each flight as it happens. We would start with only one month of data.

A good question to ask is what decision tree models would have been available to the analyst in 1987, and what kind of computers would have been used. Brieman et al. published **"Classification and Regression Trees" in 1984** and Quinlan published **"Decision Trees as Probabilistic Classifiers" in 1987**. For our purposes, the general answer is good enough. We can proceed using Decision Trees. However, we should consider that in a real life situation we should evaluate new models at every opportunity for improving a model retraining process.

SIMULATIONS

The next task is to see how that model performs on subsequent months. We built our first model on a training sample from October. We then scored the model on all the data from October, November, and December, giving us three full months of history.

THREE-MONTH RESULTS

We come back to work in January to see how we are doing. Figure 5 illustrates the plots of the monthly proportion of flights that are late, misclassified, and correctly classified as late. The first three months of model monitoring show decreasing rates of accuracy and the scatter plot shows a possible relationship between accuracy and proportion late. The plots are unspectacular but appear to show trends. The proportion of late flights and the misclassified rate are increasing. The true positive rate is decreasing. This appears to follow the theory perfectly, as data changes over time that model accuracy and performance degrades.

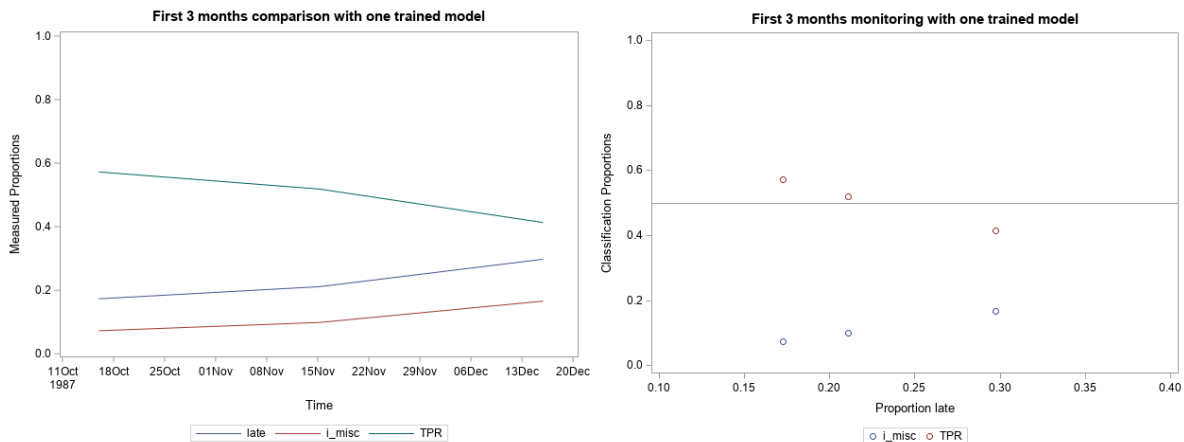


Figure 5. First Three Months of Model Monitoring and Proportion Relationship

We now have a decision to make. Should we wait another three months to see what happens? Or should we build a new model now and risk overfitting a short-term trend? We decide to do both. We will build a new model and compare the two strategies after we take a vacation and return in April.

SIX-MONTH RESULTS

We come back from our ski vacation in April 1988 to examine the results. First, we look at the results from the single model we trained based on data from October. The surprising results are shown in Figure 6. After the model decay observed in December, the model performed more accurately in months January, February, and March. This correlation with the changing proportion of late flights is marked. We can hypothesize that the pattern of late flights is different for the very busy month of December.

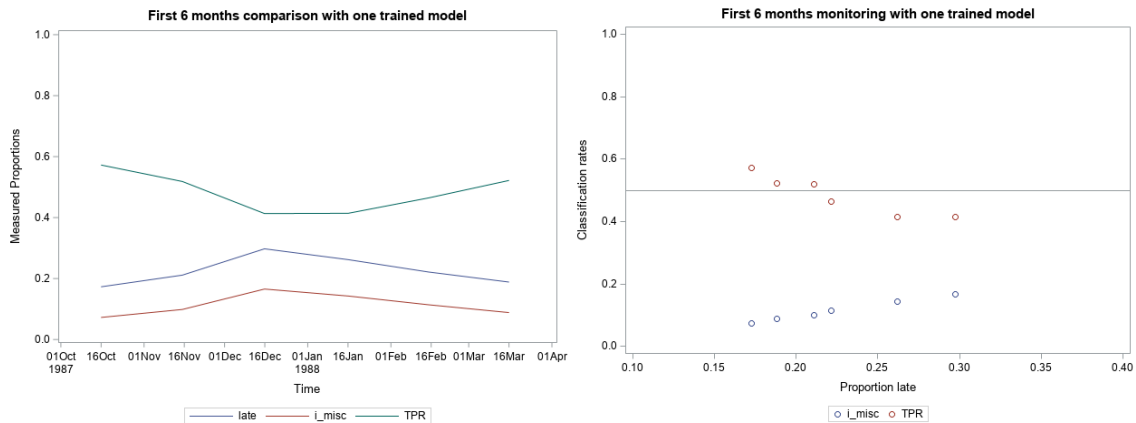


Figure 6. Six-Month Results on for the Model Originally Trained on using October Data

These scattered results are not definitive. To find a better answer, we trained and monitored models across all combinations of time periods. This included multiple-month training periods and multiple-month monitor periods for the first six months. We sampled 24 different combinations. The resulting matrix of data was fed into PROC SGLOT to create the heatmap shown in Figure 7. The color response statistic is the mean. The most accuracy monitor periods have the most training time in months. The best continuous solution across the sample is the diagonal where the most possible training months were used to create the model monitored in the subsequent month. The best discontinuous solution is the diagonal up to month 6 when three or four months of training data were better than five.

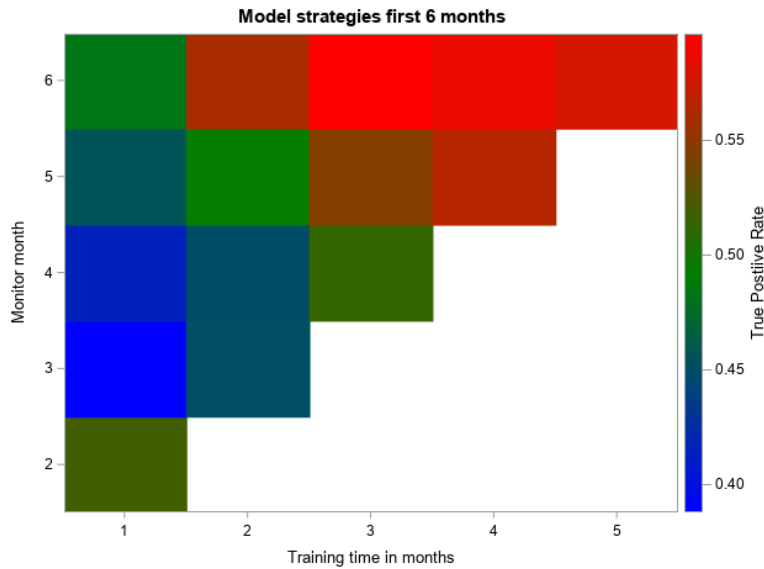


Figure 7. Model Monitoring by Training Time Period in Months

ONE YEAR RESULTS

Based on the knowledge we gained in April that longer training periods performed better on future data, we tested four strategies for the remainder of the first twelve-month period. The standard naïve single model and a model trained on each month of data are shown in Figure 8. We focus on the true positive rate as that measure that will most impact our ability to identify and react to flights that are predicted to be late. In Figure 8, the monitored true positive rate is below 50% in most months. It is surprising that the single model trained using October data is a better predictor of the next 12 months than the set of eleven models trained to predict only one month ahead for the next 12 months. However, neither model strategy is promising.

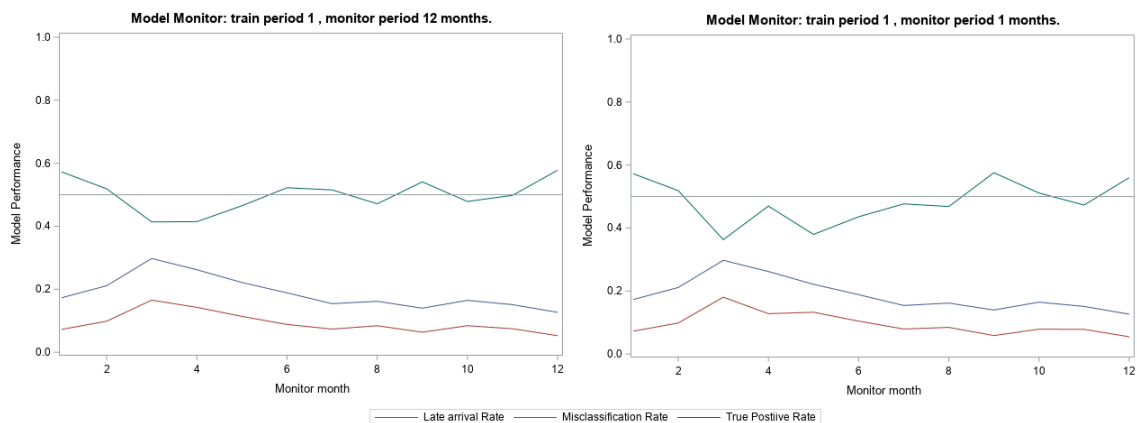


Figure 8. Model Performance of Naïve Models Trained on One Month and Each Month

Our next strategy is to test long model training periods. Each of these strategies improve performance as displayed in Figure 9. They show a much-improved true positive rate over the naïve models with true positive rates greater than 50% most of the time. In particular,

the model based on four months of training time and three months of monitoring time did very well. This is likely due to including enough data to capture periodic effects. The data is known to have seasonal patterns. There are more flights around the winter holidays and summer vacations. There are also more weather delays in the northern hemisphere in winter. However, at this point in time, October 1998, we do not have enough data to conclude that periodicity is a main effect.

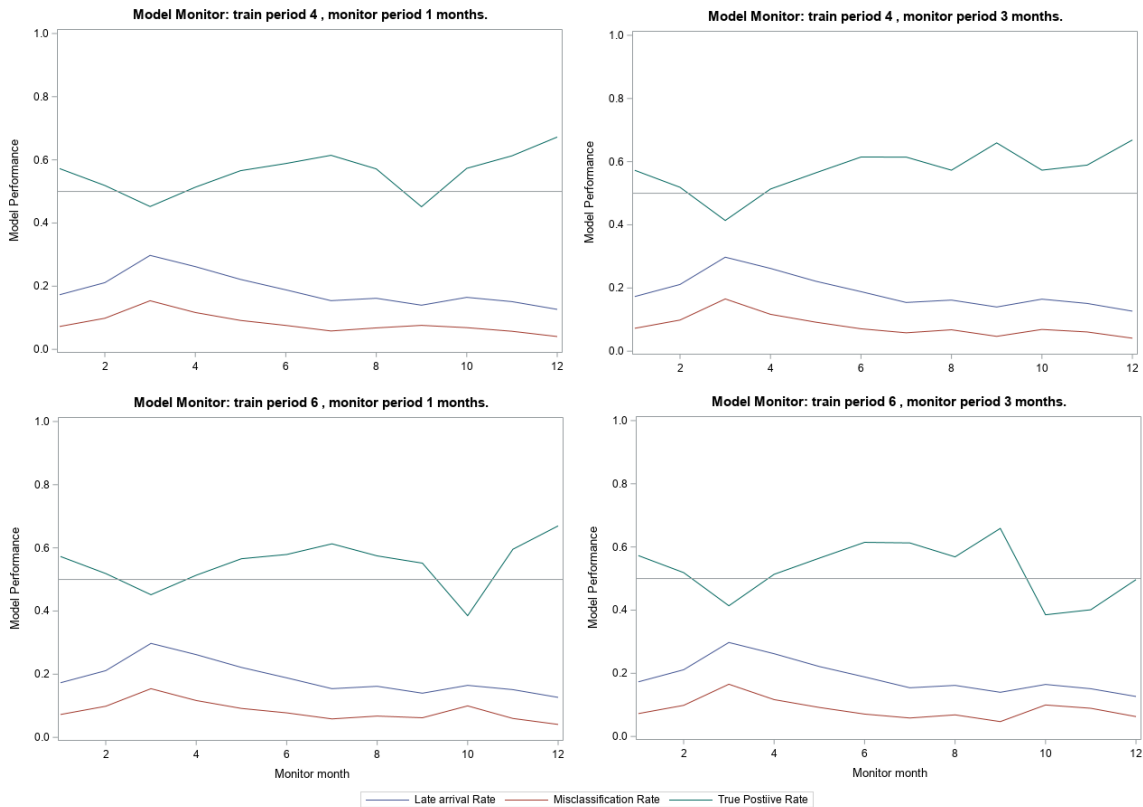


Figure 9. Four Model Training and Monitoring Scenarios with Longer Training Periods

Based on these results, we will apply the 4-3 model (4 training months to 3 monitor months) to the remainder of the data. Every three months a new model will be created using the previous four months of training data. This creates a one-month overlap in training data between consecutive models, which helps smooth the changes from one model to the next.

This strategy will result in 100 new model training events. Since we are creating both a decision tree model and a logistic regression model, that will create 200 models. Each model will be created on four months of data. Months have on average approximately 500 thousand observations and we use half the sample for training and half for testing thus resulting in training samples of approximately $500K * 4/2 = 1$ million rows, depending on the actual airline traffic for those months. The total amount of data used in training models will be approximately $200 * 500 \text{ M} = 10000 \text{ M} = 10$ billion rows!

TWENTY-FIVE YEAR RESULTS

Now jump to the beginning of the year 2013. It has been 25 years and three months since we started this project. We have been building models and monitoring their progress during that time. Before we retire from our cushy data scientist position, we will take one more look at the relative model performance of each strategy.

The single model (1-303) strategy now produces an expected result. The green top line is the true positive rate, TPR, which shows a downward trend in expected value. The blue middle line is the actual proportion of late flights, which does not show a strong long-term trend. The bottom red line is the monthly misclassification rate that shows a slight upward trend. **However, we don't** yet understand the pattern changes that cause this decay in performance; that work is outside the scope of this paper.

We have also been running the four-three strategy where each model was trained on the four most recent months of historical data and then monitored for the next three months. The difference is not as great as expected based on the first year of performance. The baseline model strategy has a mean monthly TPR of 0.382; the four-three strategy produces 0.411. Neither strategy is compelling.

Since we now have 25 years of data, we can test other long-term strategies. We believe that there are seasonal effects from monthly up to yearly if not longer. Therefore, we tested additional strategies training data on twelve and eighteen months of history. The 12-6 and 18-6 results simulations produce incremental improvements. The results are plotted in Figure 10 and listed in Table 7. Each strategy produces different cycles of better and worse model performance, and all show levels of the long-term trend to worse TPR values. Further studies might discover a strategy or model function that produces better and more reliable results.

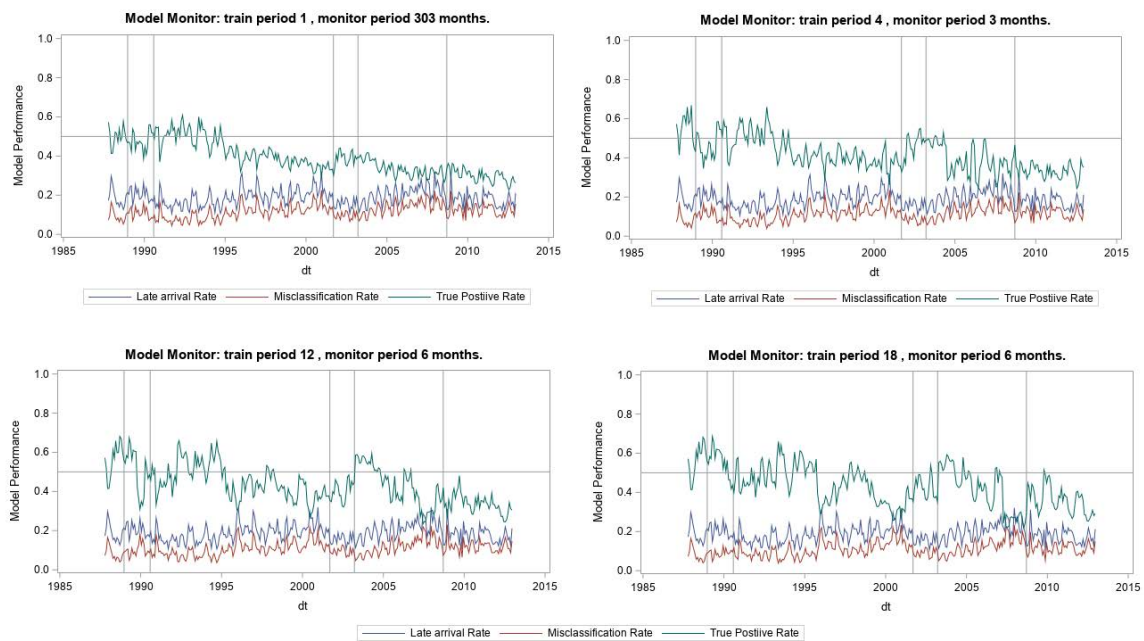


Figure 10. Long Term Model Performance of Multiple Retraining Strategies

The final column of Table 9 is the rate of months that have a true positive rate greater than 0.5. This could be an important measure of model usefulness. None of the scenarios reliably produced models with a monthly TPR greater than 0.5. This shows a weakness with

all the models used in this exercise. The last two cells are highlighted as they show a significantly elevated monthly TPR. The training fit statistics, computed on test data, indicate we may have hit the limit on core model accuracy.

Strategy	Training			Monitoring: 303 months			
	Months per model	Mean Misclassification Rate	Mean TPR	Months per model	Mean Monthly Misclassification Rate	Mean Monthly TPR	Monthly TPR > 50%
Baseline	1	0.072	0.581	303	0.117	0.399	0.134
4-3	4	0.119	0.400	3	0.113	0.411	0.155
12-12	12	0.135	0.411	12	0.111	0.422	0.207
12-6	12	0.112	0.418	6	0.110	0.428	0.249
18-6	18	0.113	0.415	6	0.111	0.427	0.270

Table 7. Comparison of Model Retraining Strategies

CORRELATION

Another aspect is correlation between model performance and the proportion of late flights as displayed clearly in Figure 11. The plots of the baseline 1-303 strategy and the best-performing 18-6 strategy are shown in Figure 11. In both cases, the misclassifications correlate well with the target variable, but the true positive rate shows significant dispersion. The 18-6 models show more true positive values above 50% especially across the greater vales of late arrival rate. This gives us more confidence in the 18-6 strategy.



Figure 11. Long Term Correlation between Classification Measures and Late Proportion

SAS MODEL MANAGER

The SAS® Model Manager product contains many of the capabilities shown in this paper. You can register all models that were creating in this exercise into a versioned repository by using a GUI application, SAS macros, Python code, or REST API services. You can execute model monitoring tasks that are similar to the ones presented here with additional capabilities. These capabilities include computing variable distributions, input and output

variable drift, and rank order statistics such as lift, captured response, KS, and Gini. SAS Model Manager can also compute a Feature Contribute Index to measure the correlation between predicted values and input variables over time. SAS Model Manager provides workflow capabilities to manage the business process shown in Figure 2. Finally, SAS Model Manager can test and deploy SAS and Python models to both batch and real-time servers for operational integration. However, SAS Model Manager does not have the extensive simulation capabilities shown in this paper. Most users are expected to be working in the moment, rather than analyzing twenty-five years of data. If you are interested in this capability, contact the author for more details.

CONCLUSION

Model monitoring and retraining are key parts of any operational model scoring process. Many paths can lead to model retraining. In this work we studied retraining models at regular intervals over a very long running process that has produced 25 years of data. The length of the time period of data used to train the models and the length of time monitoring the models in production have significant impacts on lifetime model accuracy. Data scientists should carefully monitor their models and conduct experiments to optimize those parameters.

The simulation capabilities developed for this paper were useful in testing different combinations of retraining and monitoring parameters. We found that this data contains both short and long-term periodic effects. The best combination of parameters we found used an 18-month sample to predict a 6-month interval. The core finding is that a training period should be long enough to accommodate periodic effects and should be longer than the monitoring period. We cannot generalize that specific recommendation to every process, but we want to highlight the need for observing and adjusting model retraining and monitoring. The simulation framework could be extended to test additional parameters and scenarios.

The Airline On-Time flight data from the National Bureau of Transportation Statistics continues to provide a rich source of publicly available data. The data is now complete from the 1987 through 2019.

Future work could go in several directions. We should study the effects of implementing champion-challenger strategies and dynamically changing the champion model as accuracy decreases. We should study the possibility of using forecasting to estimate when models might need retraining especially in the presence of seasonal or long-term effects. We should look at using optimization to dynamically adjust the training and monitoring parameters.

A final key finding is that the SAS system makes a great platform for importing and cleaning extremely large amounts of data, and for computationally processing that data over long periods of time. Each simulation processed billions of records over hundreds of iterations within several hours. At the end the same software was able to summarize the results and produce useful and professional tables and graphs.

REFERENCES

- Breiman, Leo, Jerome Friedman, Charles J. Stone, R.A. Olshen. 1984. *Classification and Regression Trees*. Taylor & Francis
- Quinlan, J.R., 1987. "Decision Trees as Probabilistic Classifiers." *Proceedings of the Fourth International Workshop on Machine Learning*. Pages 31-37. University of California, Irvine.
- Wicklin, Rick. 2009. "An Analysis of Airline Delays with SAS/IML® Studio." *Proceedings of the ASA Data Expo 2009*. Cary, NC: SAS Institute Inc. Available <https://support.sas.com/rnd/app/iml/papers/abstracts/airlinedelays.html>.

Bureau of Transportation Statistics. 2019. "Database Name: Airline On-Time Performance Data." Available https://www.transtats.bts.gov/Tables.asp?DB_ID=120&DB_Name=Airline%20On-Time%20Performance%20Data&DB_Short_Name=On-Time (accessed November 2019).

Duling, David. 2019. "The Aftermath What Happens After You Deploy Your Models and Decisions." *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. Available <https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3496-2019.pdf>.

ACKNOWLEDGMENTS

The author wishes to thank many people who make and support SAS software. In addition, the author thanks both Ming-Long Lam and Phil Easterling, both of SAS Institute, for their advice on both data mining and the airline industry, and Kristen Aponte for prompt editing.

RECOMMENDED READING

- [*SAS Model Manager 15.3 Users Guide*](#)
- [*SAS/STAT 15.1 User's Guide: High-Performance Procedures*](#)

CONTACT INFORMATION

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

David R Duling
SAS Institute Inc.
919 793 5663
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.

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

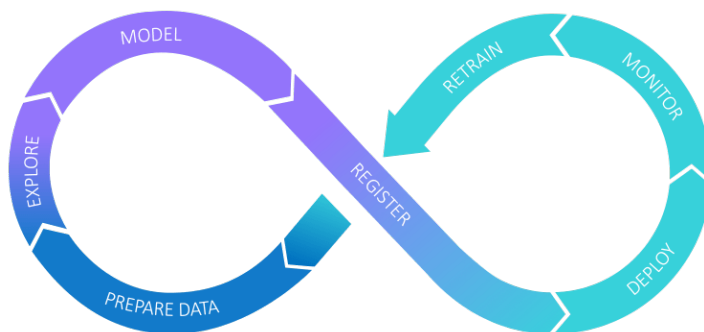


Figure 1: The model life cycle

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: pickleZip-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(validatePath, 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".

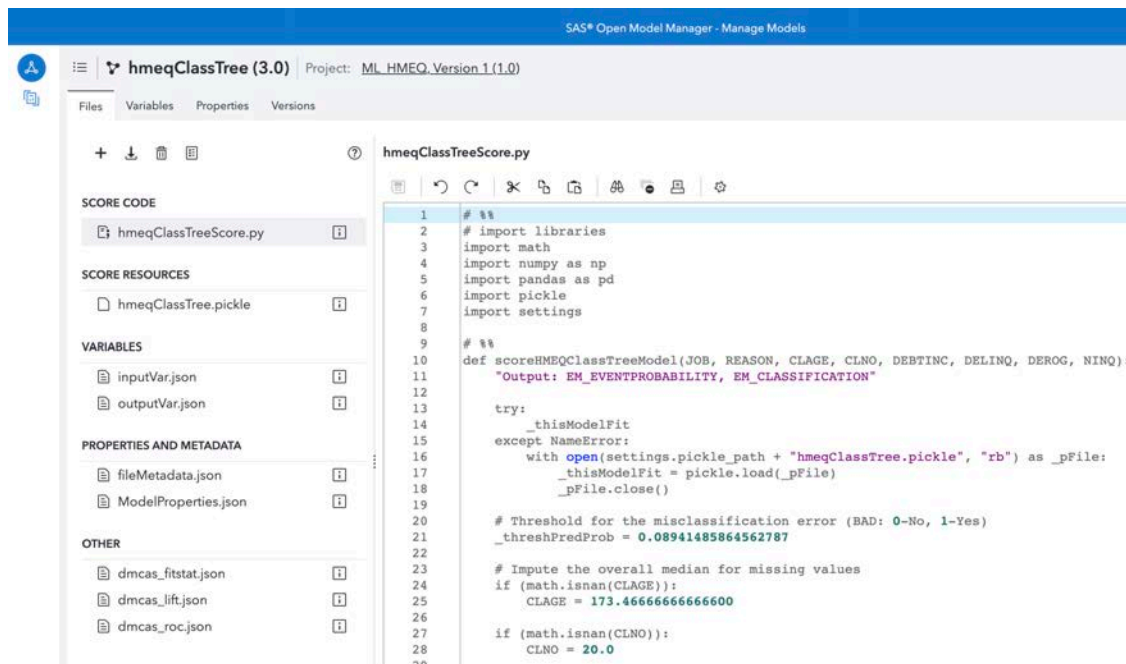


Figure 2: Imported hmeqClassTree model

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=n0b478i3vsj1pgn1dhtOctsorlet.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*.

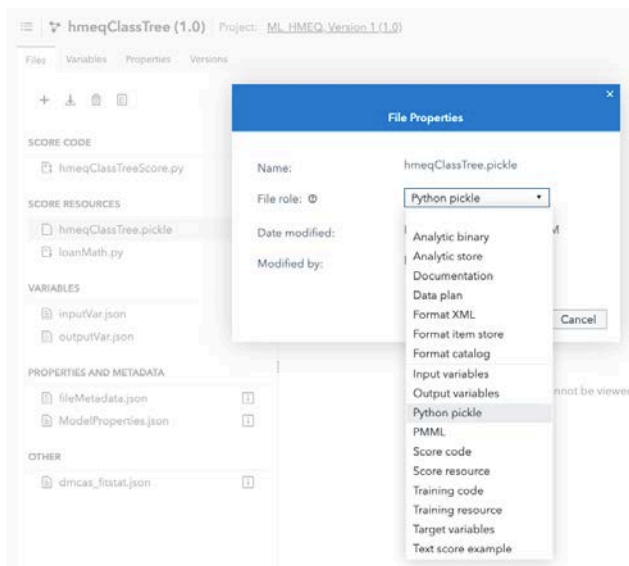


Figure 3: Setting a file type for a pickle file

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

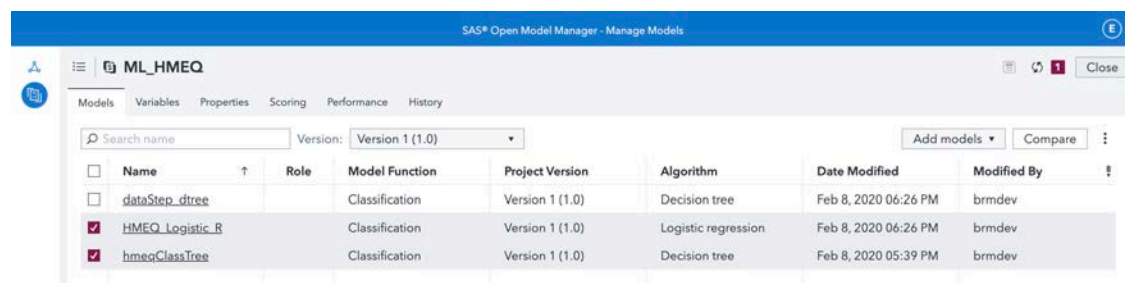


Figure 5: ML_HMEQ Project

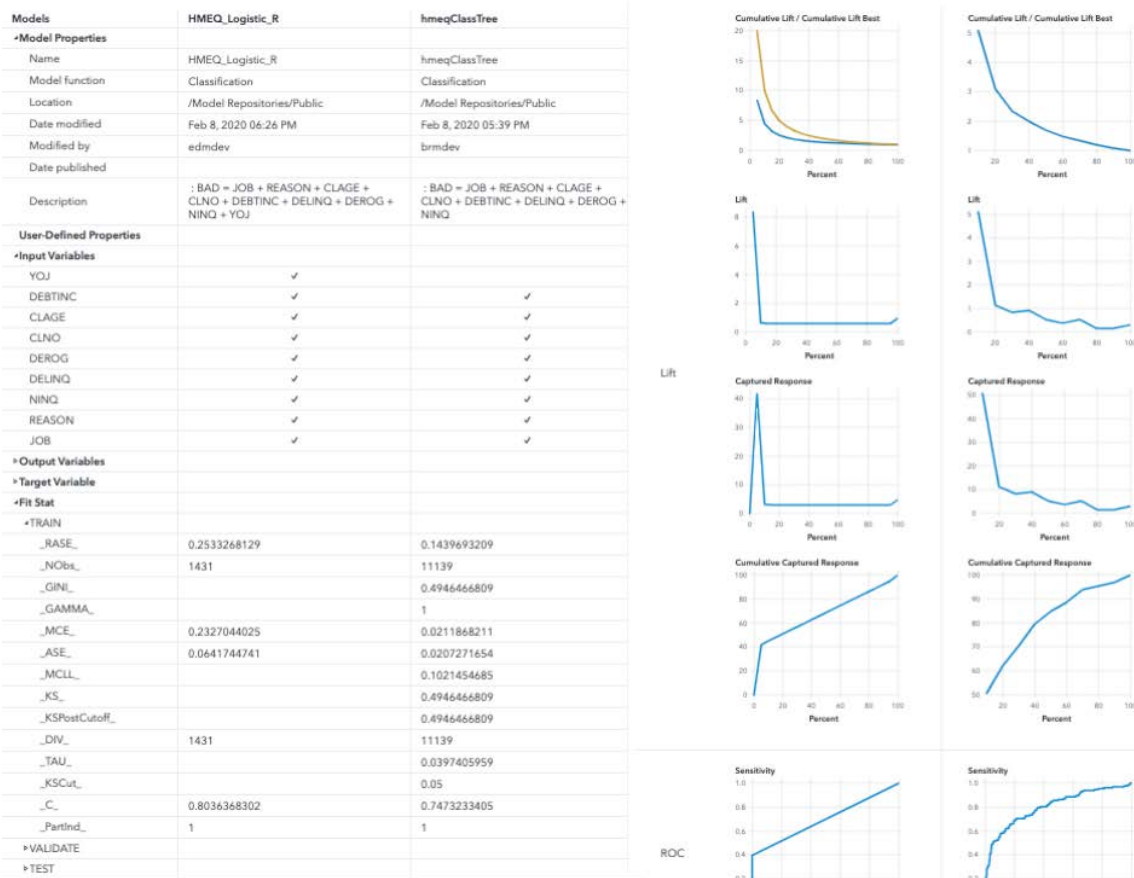


Figure 6: Model Comparison Results

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.

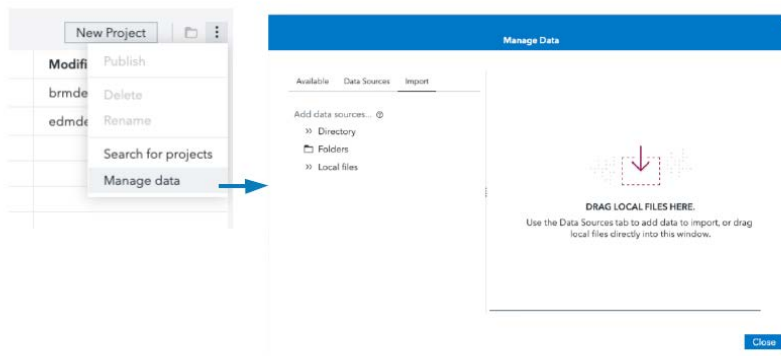


Figure 7: Managing Data

In the Manage Data dialog box, the user can import data sets, including CSV data, to be used with Model Manager’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% (2,258)	double	180.54	85.86	1.47
CLNO	1.70% (61)	double	21.35	10.19	0.17
DEBTINC	78.91% (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 sassy and swat packages are available on the SAS Software 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.

Tests Publishing Validation									
Name	Results	Status	Model N...	Project V...	Input Table	Date C... ↑	Date Co...	Created By	
<input type="checkbox"/> Test 1			hmeqClass Tree (1.0)	Version 1 (1.0)	HMEQ_JUS T1	Feb 9, 2020 04:35 PM	Feb 9, 2020 04:36 PM	brmdev	<input type="button" value="New Test"/> <input type="button" value="Run"/>

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 pyamas:
ERROR: DS2 "pyamas" 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(DEBTINC)):
    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(DEBTINC)):
        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 PRODUCTI ON 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 create a container destination descriptor and when a user requests to publish a model, the container destinations are presented alongside the other destinations.

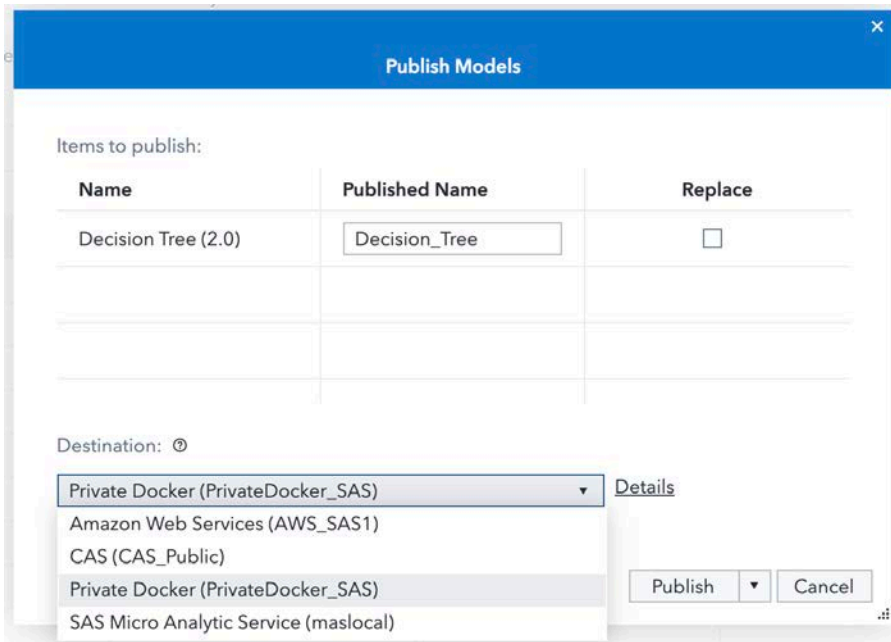


Figure 11: Model publishing to container destination

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.

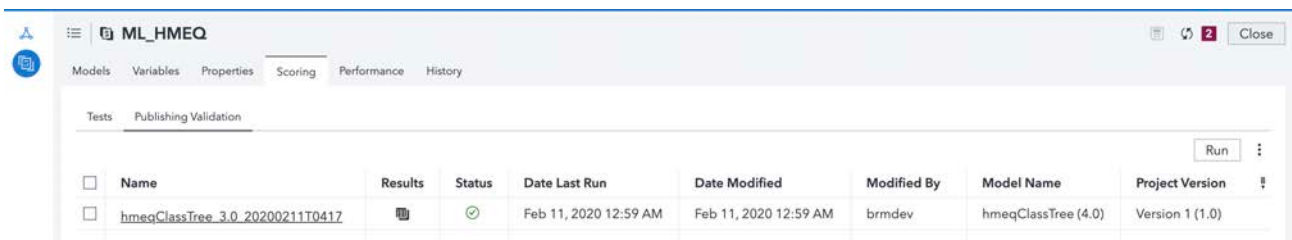


Figure 12: Publish verification results

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.

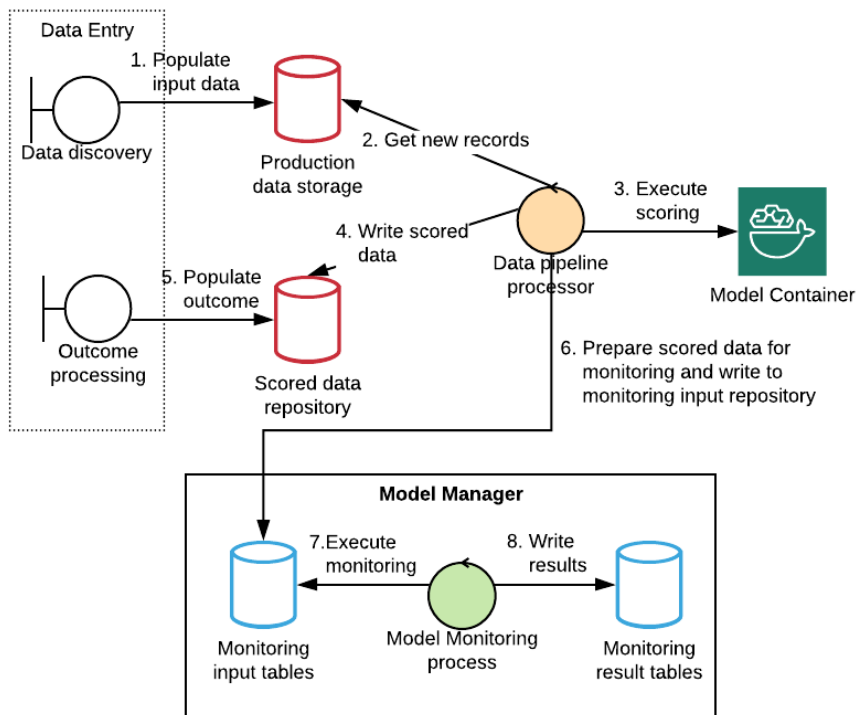


Figure 13: Example monitoring process pipeline

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".

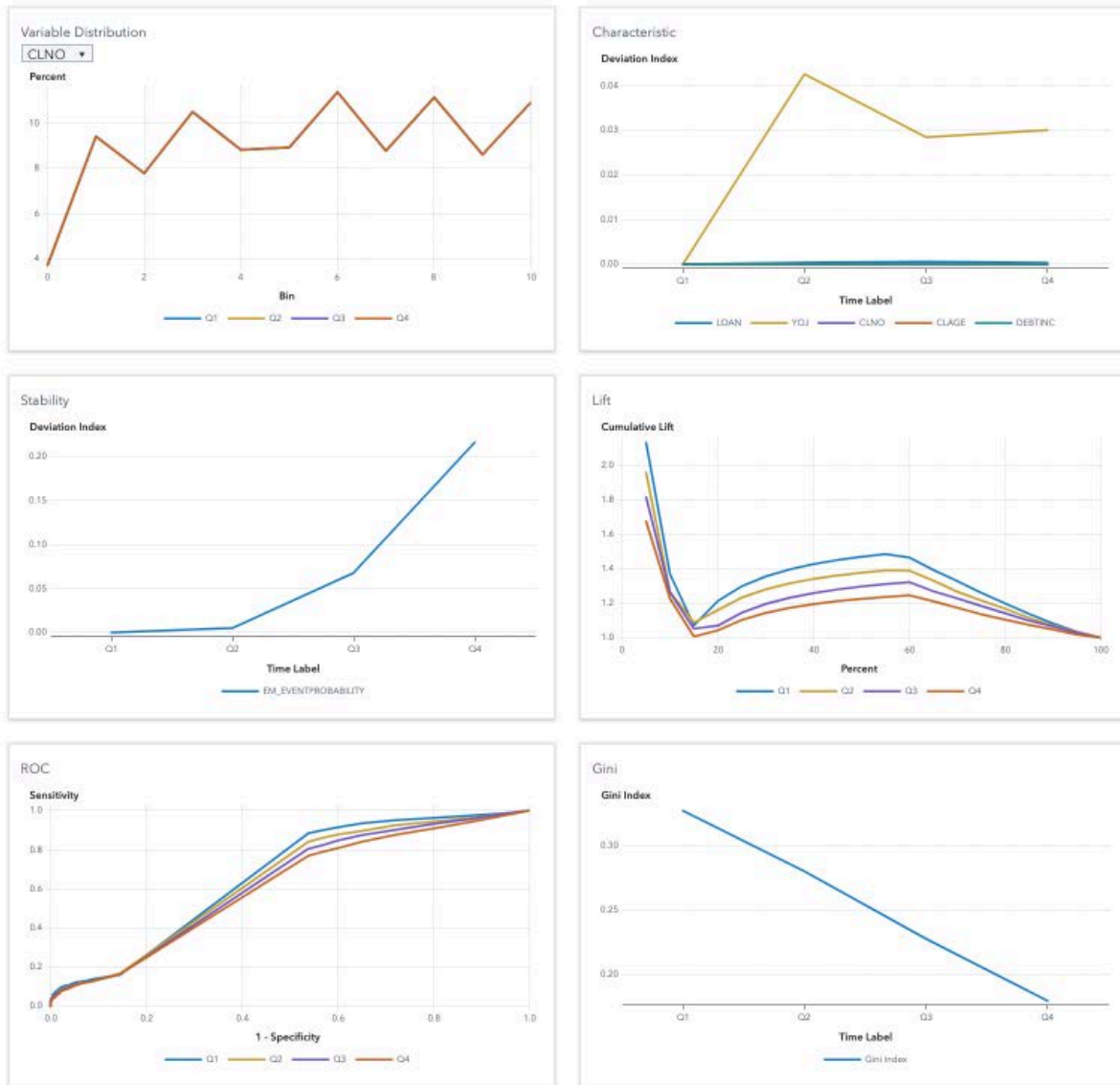


Figure 14: Performance monitoring report

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

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.

Deploying Models Using SAS® and Open Source

Jared Dean, SAS Institute Inc.

ABSTRACT

In the excitement and hype around machine learning (ML) and artificial intelligence (AI) most of the time is spend in the model building. Much less energy is expended on how to take the insights from models and deploy them efficiently to create value and improve business outcomes.

This paper will show a complete example using DevOps principals for building models and deploying them using SAS® in conjunction with opens source projects including Docker, Flask, Jenkins, Jupyter, and Python. The reference application is a recommendation engine on a web property with a global user base. This use case forces us to confront security, latency, scalability, repeatability. The paper will outline the final solution but also include some of the problems encountered along the way that informed the final solution.

INTRODUCTION

SAS Communities is a peer-to-peer community for SAS users to ask questions and find answers from each other and from experts at SAS. To improve the experience of users on the site we wanted to create personalized recommendations. To help visitors find articles of interest among the tens of thousands of active articles on the site. In Figure 1 Screen Shot of Recommendations you can see the finished product. I am the first to acknowledge that this does not make the most exciting demo but there is coordination between different personas and technologies to make the recommendation experience transparent to the end user.

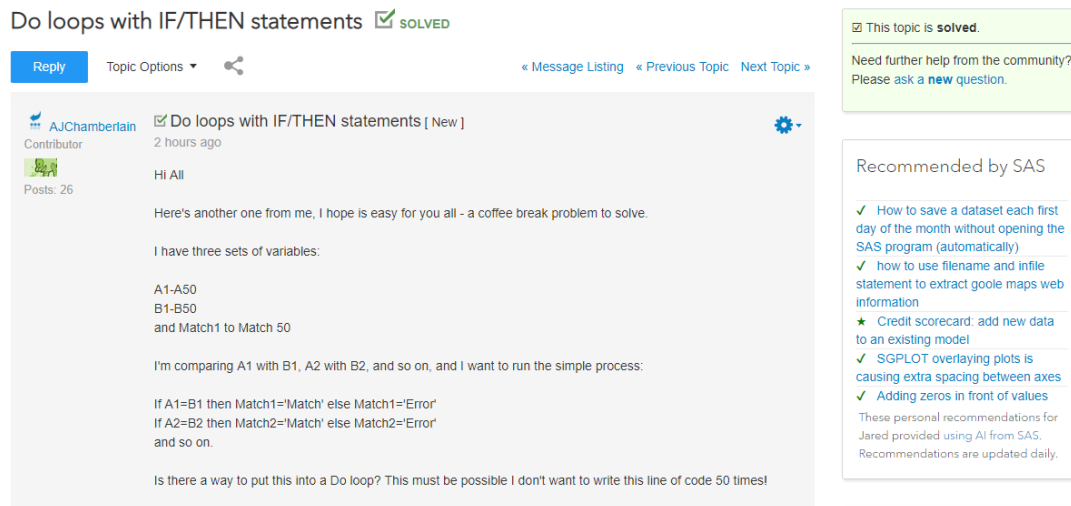


Figure 1 Screen Shot of Recommendations

My goals in writing this paper are: first is to demonstrate how SAS is using its technology to solve its own business challenges of improving the user experience on our SAS Communities web property. Second, to provide a template for how you can implement these ideas in your organization either a web property or some other service that would benefit from recommendations. Next, to highlight how SAS works with open source tools. Finally, how SAS can be used in a cloud ready environment without long installation and configuration times.

From the software point of view, this project uses several SAS products along with open source tools. SAS and open source tools work in concert with each other to help users be more productive and impactful to their organizations.

From the human capital perspective, this project involved several personas. A data scientist to prepare the data and build the models, a systems engineer to deploy and monitor the model on a daily basis, a security engineer to help ensure compliance and safety of our applications, a full stack developer to integrate the API call into the website.

ARCHITECTURE AND TOOLS

This project uses SAS DATA step for the data prep and the FACTMAC procedure, as part of SAS Visual Data Mining and Machine Learning (VDDML) for the recommendation model.

Table 1. Utilized Open Source Tools is a list of the Open source tools used across the project. If you're a SAS user, then many or even all the tools in the table might be unfamiliar. If your first impression is that the table is rather long, I agree. Integration with open source tools will bring you into contact with many projects all with different levels of maturity and style and function. Don't be intimidated, keep in mind this project spans several personas and many of these tools are standard and well known across IT / systems professionals. It would be very unusual for a single person to use all these tools.

Table 1. Utilized Open Source Tools

Open Source Tool	Description
Jupyter	Web Based IDE; supports many kernels including Python and SAS.
SASPy	Open Source project that allows python programmers to use the SAS computing engine (9.4m0 or newer).
Python-SWAT	Open Source project that allows python programmers to use the SAS Viya computing engine.
Docker	Container technology that allows quick consistent deployment of environments.
Kubernetes	Open source project to manage and scale docker containers.
OpenStack	Private cloud infrastructure.
Terraform	Orchestration tool for quickly deploying cloud assets.
Git	Source management tool.
Python	Open Source programming language; object oriented; created in 1985.

Analytics professions have been building models for years and years. It is the fun and exciting part of the job for most nerds like me. The part that is often manual time consuming and costly to the organization. This paper demonstrates how SAS can be used, along with open source tools, to create a modeling and deployment process that follows the Continuous Integration/ Continuous Deployment (CI/CD) methodology

Figure 2. SAS Community Recommendation Process illustrates the process that I have used to create personalized recommendations for the SAS Community users based on their past browsing history. The Build box from Figure 2 is the domain of the modeler and you can read more about that role in this project in the Modeling section.

While I had the main responsibility for this project, no successful project is done alone. Partnering with colleagues within your organization and leveraging their strengths and expertise will reduce implementation time and cost to the organization.

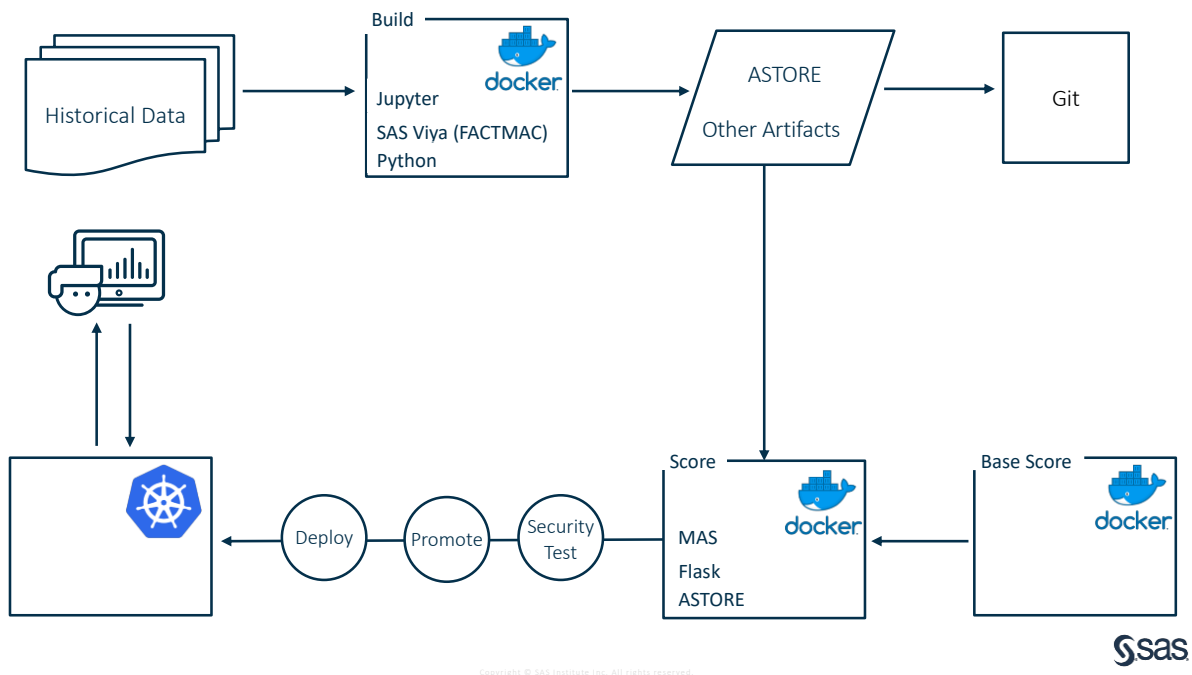


Figure 2. SAS Community Recommendation Process

The

MODELING

This is primary domain for the data scientist (or appropriate vogue business title). They have three main responsibilities: prepare the data, build a great model, generate the artifacts for packaging and deployment.

PREPARING THE DATA

There are several required steps in preparing the data for modeling. The size of the and frequency of this problem posed several challenges in this area. The first challenge is the data size. SAS Communities is a very popular site and the transactional historical data is large. Using data from late 2015 to present is about 40GB of data. So the first task is to remove the variables and rows that do not relate to our modeling problem.

The next challenge is creating ratings. In most cases users provide recommendations on the products or services they consume. Netflix ratings on movies is one example in this case we do not have recommendations, but machine learning techniques are not very effective if there is no discrimination between articles. So, we must augment the data using a process called implicit feedback in which we will take all the articles a user has seen and then randomly sample from a set of articles they have not seen to get two equal sized partitions. The viewed articles receive a rating of 1 and the non-viewed articles receive a rating of 0. The coding challenge is to extract the articles viewed by a particular user and then sample without replacement from non-viewed articles in an efficient manner. The code must be as efficient as possible because there are over 130,000 registered members of the SAS Community and this process is run every day to provide the most up to date recommendations. To illustrate the speed requirement if the process took just one second per user that would be more than 36 hours if done serially so parallelization and efficiency are paramount.

In using SAS 9.4m6 I was able to reduce the data prep to 23 minutes through macros and hash tables. I then moved the DATA step code to CAS DATA step and the time was reduced to just over 3 minutes on the same hardware. CAS DATA step has the added benefit in this case that when the code moved to a cloud deployment it maximizes the computing for whatever size system it runs on without additional code modifications on my part.

Here is my code for implicit feedback per user:

```
data mycas.implicit_feedback;
  array found[&nconv.] _temporary;
  array compressed[&nconv.] _temporary;
  set mycas.conversations;
  by user_uid;
  retain count;
  if first.user_uid then do;
    count=0;
    do i = 1 to dim(found);
      found[i]=0;
    end;
  end;
  found[conversation_ord] = 1;
  rating=1;
  count+1;
  output;
```

The code above initializes two temporary arrays to store the conversations a specific user has viewed from the universe of all conversations. The size of the array &nconv is the number of unique conversations at the time of modeling. Every article is output with a rating of 1.

The code below runs after we have a complete list of the conversations viewed by a user. It randomly selects a set of conversations that a user has not viewed and sets their rating to 0. The resulting dataset, mycas.implicit_feedback, has exactly twice as many observations as the starting dataset, mycas.conversations.

```
if last.user_uid then do;
  unseen = 0;
  seen = 0;
  /* make the compressed array of unseen conversations */
  do i = 1 to dim(found);
    if found[i]=0 then do;
      unseen+1;
```

```

        compressed[unseen] = i;
    end;
    else do;
        conversation_ord = i;
        seen+1;
    end;
end;
/* shuffle them */
retsize = min(count,max(seen,unseen));
if seen > unseen then put user_uid= seen= unseen=;
do i = 1 to retsize;
    j = floor(rand('uniform')*unseen)+1;
    temp = compressed[i];
    compressed[i] = compressed[j];
    compressed[j] = temp;
end;
do i = 1 to retsize;
    conversation_ord = compressed[i];
    rating = 0;
    output;
end;
end;
run;

```

BUILDING THE BEST MODEL

With the data prepared we can move to modeling. Part of the business requirements for the recommendations were to create recommendations that favor articles with accepted solutions and favor newer articles. The logic for this is that accepted solutions will be more useful to users and the articles you've viewed recently are more applicable to your current interests than those from several years ago. The FACTMAC procedure doesn't have a weight statement yet so I created duplicate rows to nudge PROC FACTMAC to follow these requirements.

Here is the SAS code to weight data by both recency and having an accepted solution:

```

%let wf=3;
data mycas.weighted_factmac / single=no;
    set mycas.implicit_feedback;
    if _n_=1 then do;
        declare hash conv (dataset:'mycas.conversation_uid_index');
        conv.DefineKey('conversation_ord');
        conv.DefineData('start_id', 'end_id');
        conv.DefineDone();

        declare hash row_lookup (dataset: "mycas.conversations");
        row_lookup.DefineKey('id');
        row_lookup.DefineData('event_time_ms', 'isSolvedTopic');
        row_lookup.DefineDone();
    end;
    if rating=0 then do;
        conv.find();
        id = (start_id + floor((1+end_id-start_id)*rand("uniform")));
        row_lookup.find();
    end;
    decay = (datepart(event_time_ms)-&mindate.)/(&maxdate. - &mindate.);

```

```

rep = int((1-decay)*10);
daysback = intck("DAYS", datepart(event_time_ms), today(), "C");
do i = 1 to rep*&wf.;
    output;
end;
run;

```

Using HASH objects I can quickly look up the information needed to properly weight the observations.

One important feature of FACTMAC is the autotuning. The AUTOTUNE feature uses optimization to search for the best hyperparameters saving me time hunting for the ideal combination of settings (that could change over time). The running of this model takes hundreds of CPU hours to complete the search for the best hyperparameters but it saves much more than that in human capital costs. For more information on AUTOTUNE see the suggested reading section.

In the latest iteration, FACTMAC modeling using the AUTOTUNE statement costs about 300 CPU hours of time and because the problem is only changing slightly each day (one new day of data among more than 1000 days) I reduced the daily run time by using some of the options available in AUTOTUNE. Here is my code for the FACTMAC procedure:

```

proc factmac data=mycas.weighted_factmac outmodel=mycas.factors_out;
    autotune maxtime=3600 objective=MSE
        TUNINGPARAMETERS=(nfactors(init=20) maxiter(init=200)
learnstep(init=0.001));
    input user_uid conversation_uid /level=nominal;
    target rating /level=interval;
    savestate rstore=mycas.sascomm_rstore;
run;

```

I use the MAXTIME option to limit the search to one hour and I use the TUNINPARAMETERS option to start with best configuration from my last complete run (which I run periodically). This strategy gives me the best known hyperparameters within a time budget and an opportunity to find even better hyperparameters with the extra time.

CREATING ARTIFACTS

With the data prep and modeling complete I can now create all the needed artifacts to quickly and efficiently deploy a scoring model. The main artifact I need is an ASTORE (see suggested reading for more information) which is a portable compressed binary object that contains the scoring logic to predict how much a specific user would enjoy a specific conversation on the SAS Community. In addition to the ASTORE, Table 2. Modeling Artifacts details the items created.

Table 2. Modeling Artifacts

Artifact	Purpose
Data set of active articles	To speed the scoring with a preloaded list
Json file of most popular articles	This is the fallback recommendation when nothing better can be presented

Artifact	Purpose
Articles viewed in the last N days	We don't want to continue to recommend the same things over and over so if you've looked at an article we won't recommend it for a while.
Data file with keys and timestamps	This file is used to ensure files are transferred correctly and for reporting information at score time.
FACTMAC ASTORE	Compressed binary container the scoring logic for each user and item.

The artifacts from Table 2. Modeling Artifacts are produced daily because we get updates to the data daily. These artifacts are packaged into a Docker container they could also be placed under source management in Git.

ORCHESTRATION

The orchestration of this process has evolved over time. It is my recommendation that you not try to automate and script everything in the beginning but instead continuously improve the process over time in progressive steps. This is one of the main philosophies of CI/CD that I think is well aligned with how projects develop and mature over time.

This project started out with just a Jupyter notebook to create the artifacts. Figure 3 Improvement of Process to reach Cloud Ready show the evolution toward fully cloud deployed and autonomous. Here are the written details:

After the initial model was created, I encapsulated the notebook in a Docker container to ensure software stability from run to run. After that, I created a bash script to run the steps via the Unix utility cron instead of running the steps manually each day. I was only ready for cron once the process was repeatable and stable. I expected each step to work, I was not hoping it would work. With the bash script I could automate the process to run daily even when I was out of the office. Over time, I needed better monitoring. I moved my bash script into Jenkins, an open source orchestration tool use by SAS R&D. After a few weeks it was decided the project should be managed by IT not R&D so I needed to migrate my script from Jenkins to Bamboo (a comparable product from Atlassian) which was the orchestration tool of choice for the IT team. When I moved the project to Bamboo, we discovered that the bamboo agents (servers available for tasks) were not sufficiently large to run my project. So, I evolved the project one last time to initiate the hardware resources in OpenStack (an open source cloud operating system) using Terraform.

I share these details with you to demonstrate that the result of a fully scalable cloud deployment of SAS to prepare, model, and score was not the initial release instead it was an evolution to meet the project objectives. The project can now access large amounts of computing power for a short period of time and upon completion release the computing power to others.

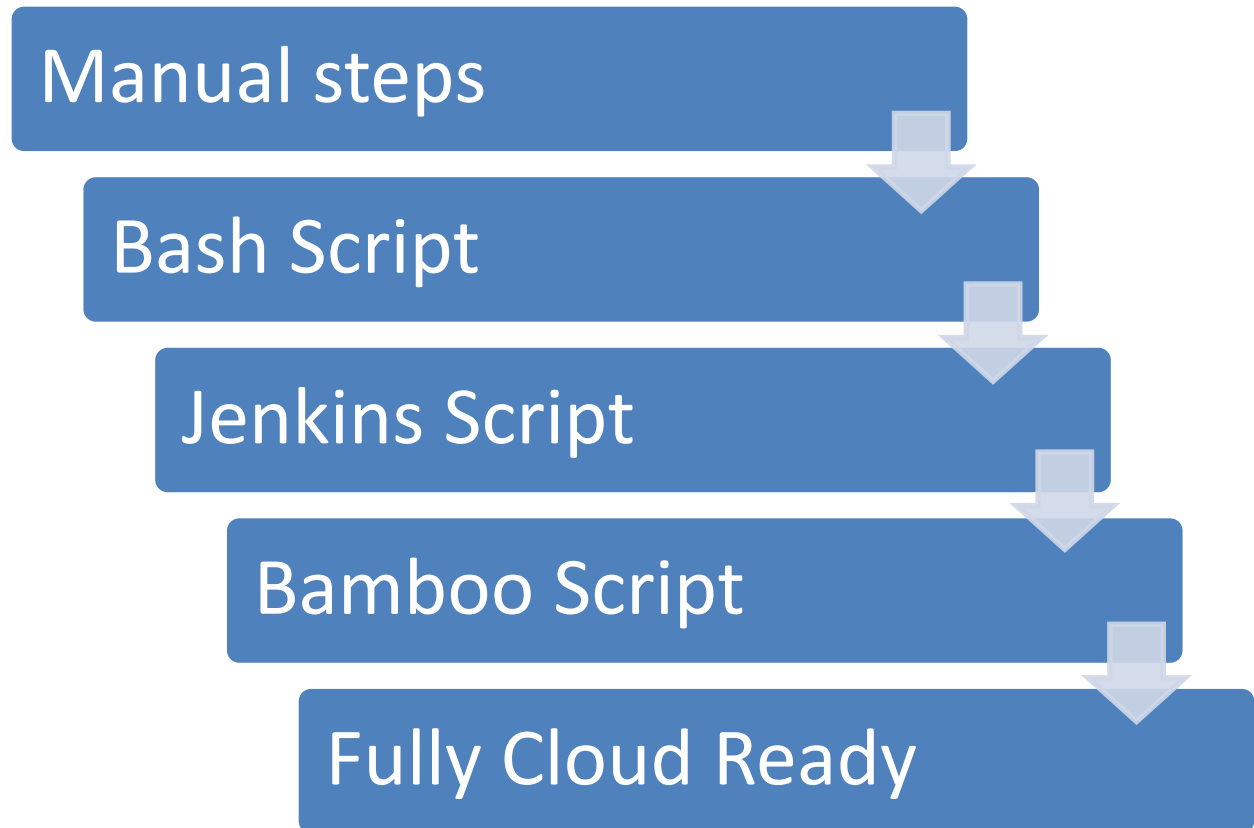


Figure 3 Improvement of Process to reach Cloud Ready

DEPLOYMENT

The deployment aspect of this project was the newest to me. Because of the need to run daily, a robust automated deployment process without an outage was required. We wanted users to have personalized recommendations with less than a two second delay. Working with the IT staff, the best practice is to build a scoring container in docker and then manage that container via Kubernetes. Docker was also used to build the model artifacts because it guaranteed an immutable environment and portability to different hosts.

The scoring docker container is different from the modeling building container in a few significant ways. The first difference is that the scoring container is purpose built to score just one model, in this case, personalized recommendations for SAS Community users. The model building container has a complete version of SAS along with visualization tools. This makes the sizes very different. The scoring container is around 800MB and the model building container is around 13 GB. The smaller container makes it easier to deploy and scale. This smaller container also adds a measure of security because the scoring container is exposed to the internet (with several security measures) but has only the essential components needed to score incoming requests not a general-purpose workflow.

Inside the scoring container we have a Flask application running under Gunicorn. These are both open source projects written in Python. The Flask application receives the incoming API request and sends it to the ASTORE through the micro analytics service (MAS). MAS is a capability through SAS Model Manager or SAS Decision Manager. By using MAS, you can achieve very low latency scoring. When a user hits the SAS Community webpage while logged in, a request is sent to Kubernetes cluster which is running the scoring docker

containers. It is then routed to a specific container and received by the flask application. The flask application sends the userid and the number of items requested N (the default is 5; max is 25). To return the best N article recommendations, all active articles must be scored (around 85,000 minus any articles that the user has viewed in the cooling off period). After all the appropriate articles have been scored, it sorts the list by the users predicted rating and takes the top N . This typically happens in about 200ms which means it is performing about 400,000 transactions per second per core after initial startup. The actual flow of information is relatively simple. A request is made from a user by viewing a SAS Community webpage and the request is validated by the firewall and routed to a scoring container running in Kubernetes (known as a pod) the scoring container processes the request and then returns the top n items back as a json string to the web page all in about 200ms. See Figure 4. Scoring Infrastructure.

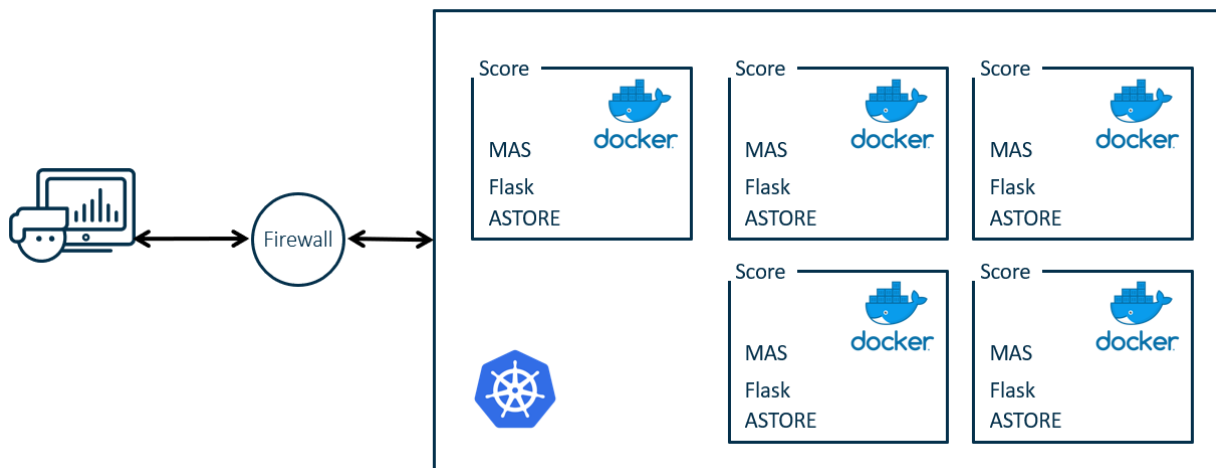


Figure 4. Scoring Infrastructure

CONCLUSION

This paper has detailed how SAS created personalized recommendations for the SAS Communities web property using SAS technology along with open source tools. This project uses a CI/CD framework that allows for automated daily updates to recommendations so that users are always getting the most current recommendations possible.

The recommendations are using cloud infrastructure to eliminate persistent hardware requirements for building the daily model and it can be scaled to meet the desired time constraints. The runtime scoring is using standard IT tools, Docker and Kubernetes, to fit seamlessly into workflows.

The SAS system integrates with open source tools to provide leading edge analytics in a cloud ready environment.

REFERENCES

Laster, Brent "What is CI/CD?" opensource.com 06 Aug 2018 Accessed February 22, 2019
<https://opensource.com/article/18/8/what-cicd>

Dean, Jared. "How to Build a Recommendation Engine Using SAS® Viya®." Proceedings fo the SAS Global 2018 Conference, Denver, CO. Available at <https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2095-2018.pdf>

SAS Micro Analytics Service

<https://go.documentation.sas.com/?docsetId=masag&docsetTarget=titlepage.htm&docsetVersion=5.2&locale=en>

AUTOTUNE statement

https://go.documentation.sas.com/?docsetId=casml&docsetTarget=viyaml_introcom_sect005.htm&docsetVersion=8.11&locale=en

FACTMAC procedure

https://go.documentation.sas.com/?docsetId=casml&docsetTarget=viyaml_factmac_toc.htm&docsetVersion=8.11&locale=en

ACKNOWLEDGMENTS

The author would like to thank Chris Bailey, Paul Kent, Jorge Silva, Brett Smith, Tom Weber, and Brett Wujek for their contributions to the paper. He is also grateful to <<>> for their editorial contributions

RECOMMENDED READING

- What is CI/CD?
- How to Build a Recommendation Engine Using SAS® Viya®.
- *AUTOTUNE*
- *FACTMAC*
- *MAS*

CONTACT INFORMATION

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

Jared Dean
SAS
Jared.Dean at SAS dot 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.

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.

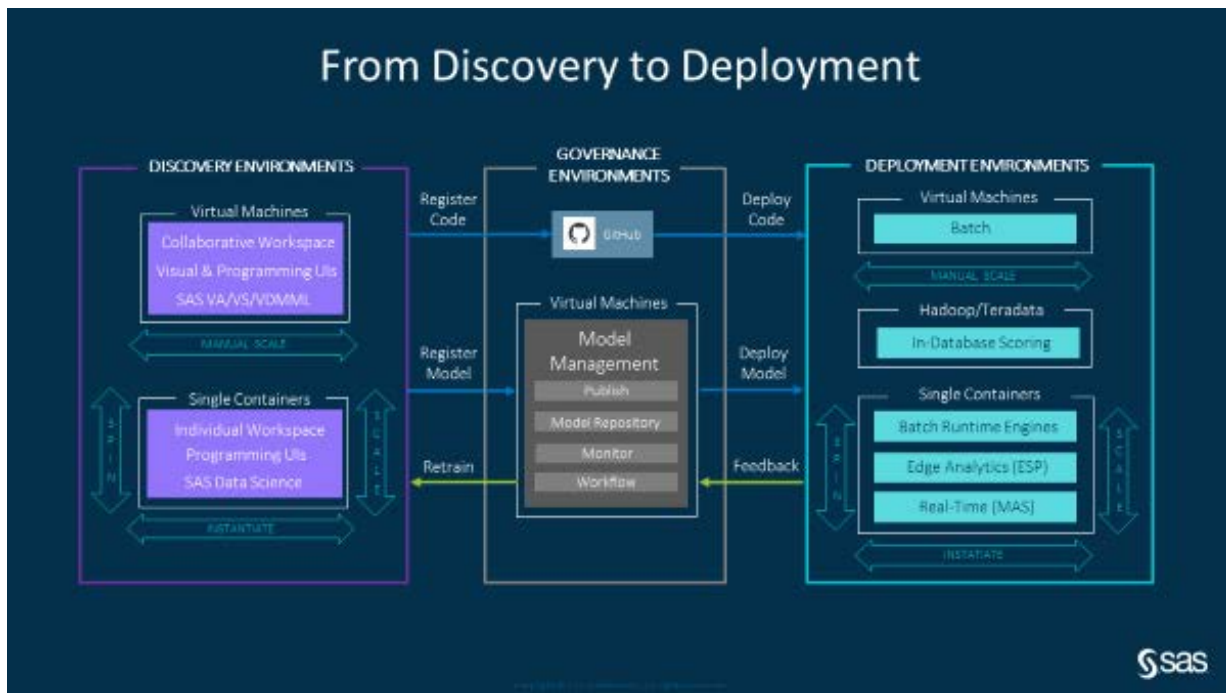


Figure 1. Analytical Lifecycle

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."**

[\(https://thenewstack.io/pets-and-cattle-symbolize-servers-so-what-does-that-make-containers-chickens/\)](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

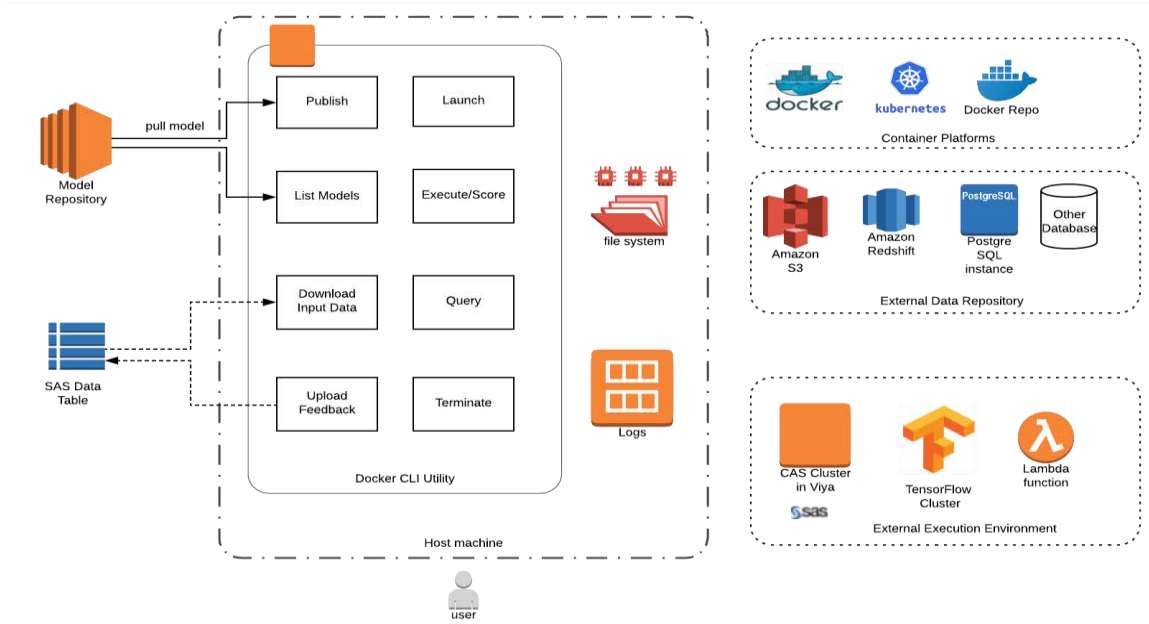


Figure 2. Python Utility

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.

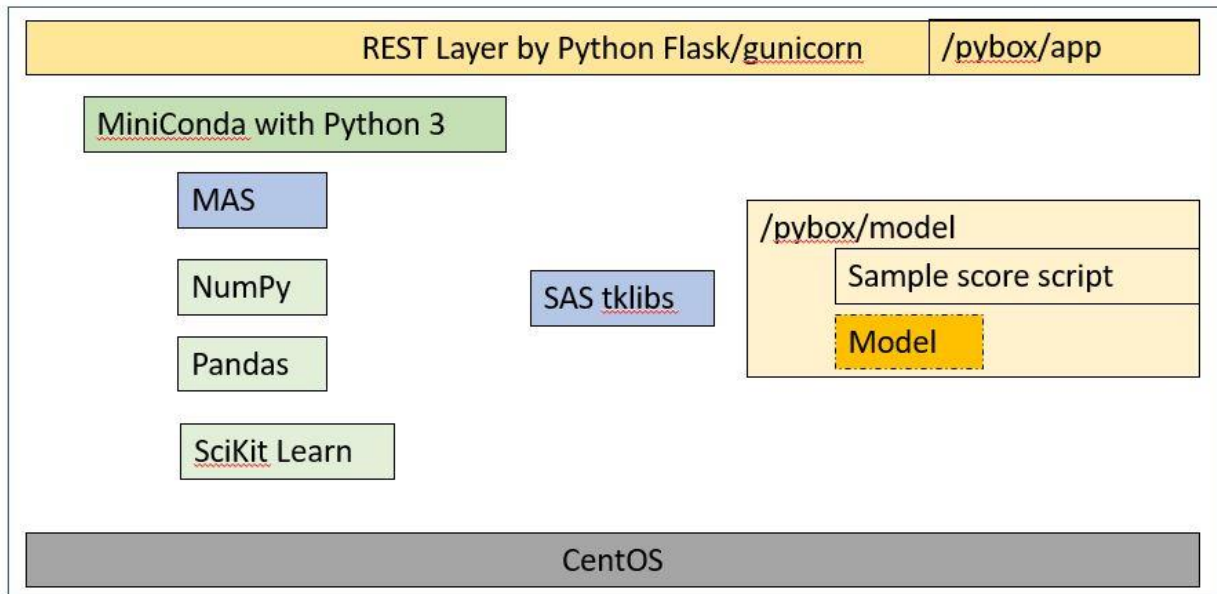


Figure 3. MAS Base Image Structure

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-a877-39249d2a98ab
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-05,1.000038,0,3.431873e-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
5951e-05,1.000038,
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]:
```

	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]:
```

	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 x 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))
```

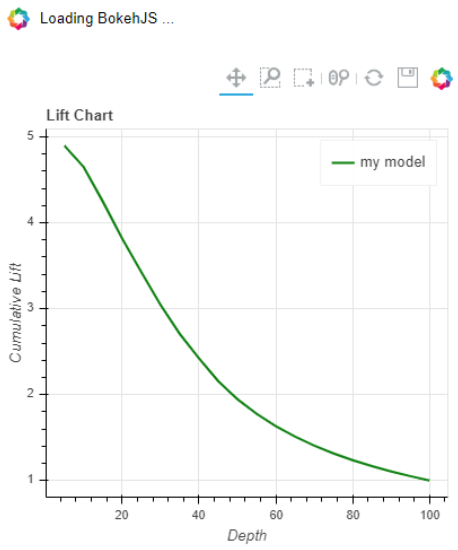


Figure 13. Generate Lift Chart

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	0.659
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	0.659
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	0.659
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	0.659
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	0.659
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	0.663
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	0.669
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	0.675
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 x 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.

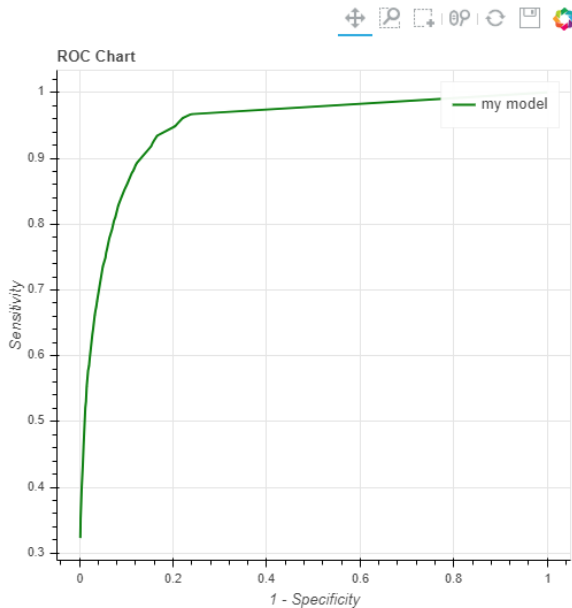


Figure 15. Generate ROC chart

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.
```

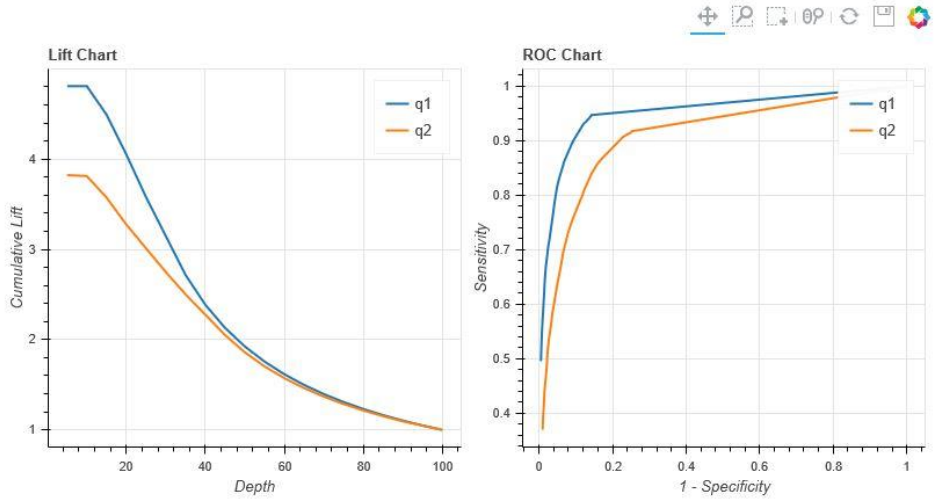


Figure 16. Side by Side Compare the Scoring Results

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
<input type="checkbox"/>	mm-docker-models-eks-worker-nodes	2019-02-14 10:08:09 UTC-0500	CREATE_COMPLETE	NOT_CHECKED	Amazon EKS - Node Group
<input type="checkbox"/>	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-43d3-9e8d-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: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.

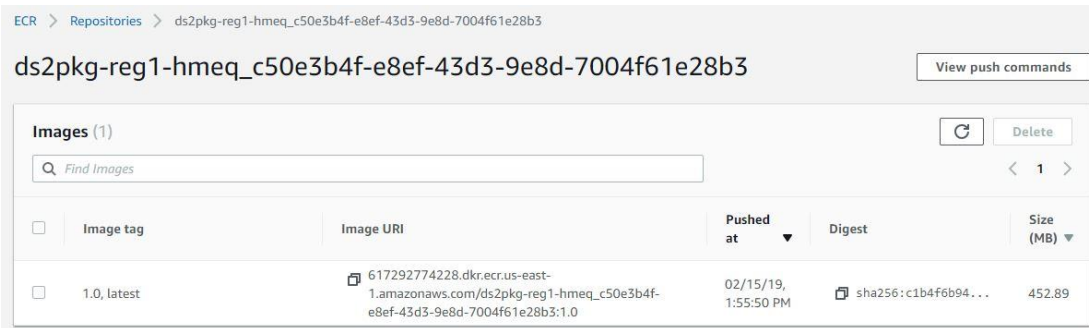


Figure 21. AWS – Use Elastic Container Registry (ECR) to Verify Results

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

```
In [4]: launch("617292774228.dkr.ecr.us-east-1.amazonaws.com/ds2pkg-reg1-hmeq_c50e3b4f-e8ef-43d3-9e8d-7004f61e28b3")

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.

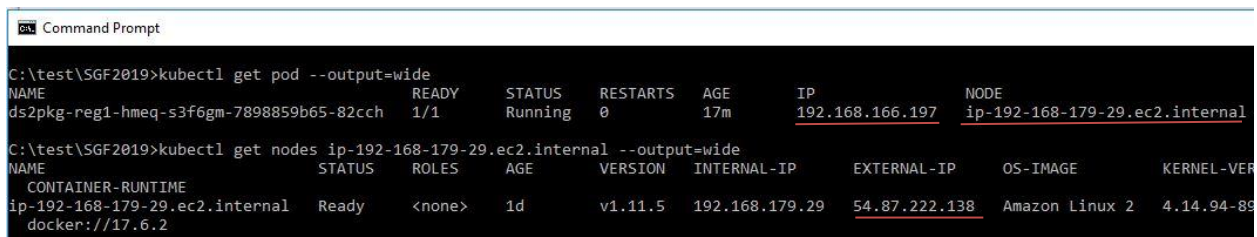


Figure 23. AWS – Verify Information with kubectl

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_mmm-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:latest
=====
```

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_d00bb4e3-0672-4e9a-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.

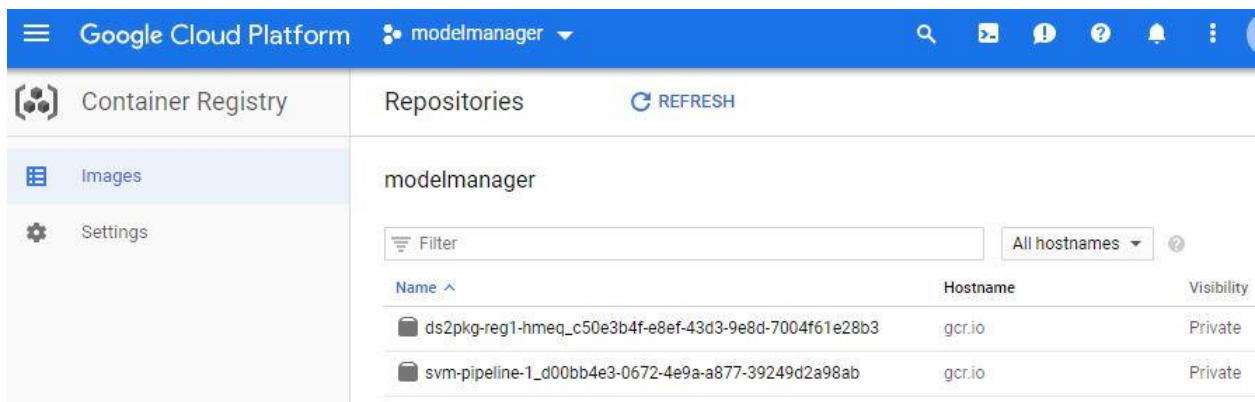


Figure 28. GCP – Use Google Cloud Platform Console to Verify Results

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.

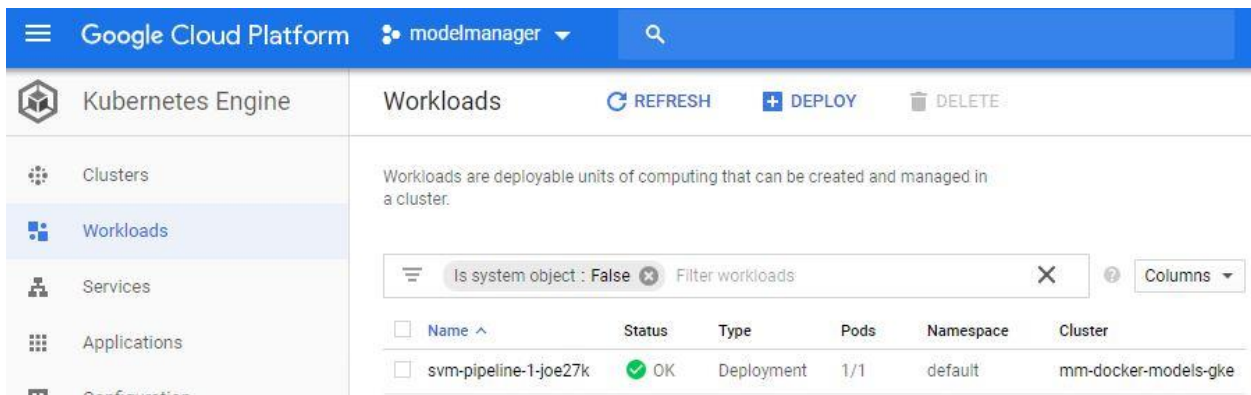


Figure 30. GCP – Verify Deployment in Google Kubernetes Engine Workloads

Figure 31 shows an example of verifying the service pod.

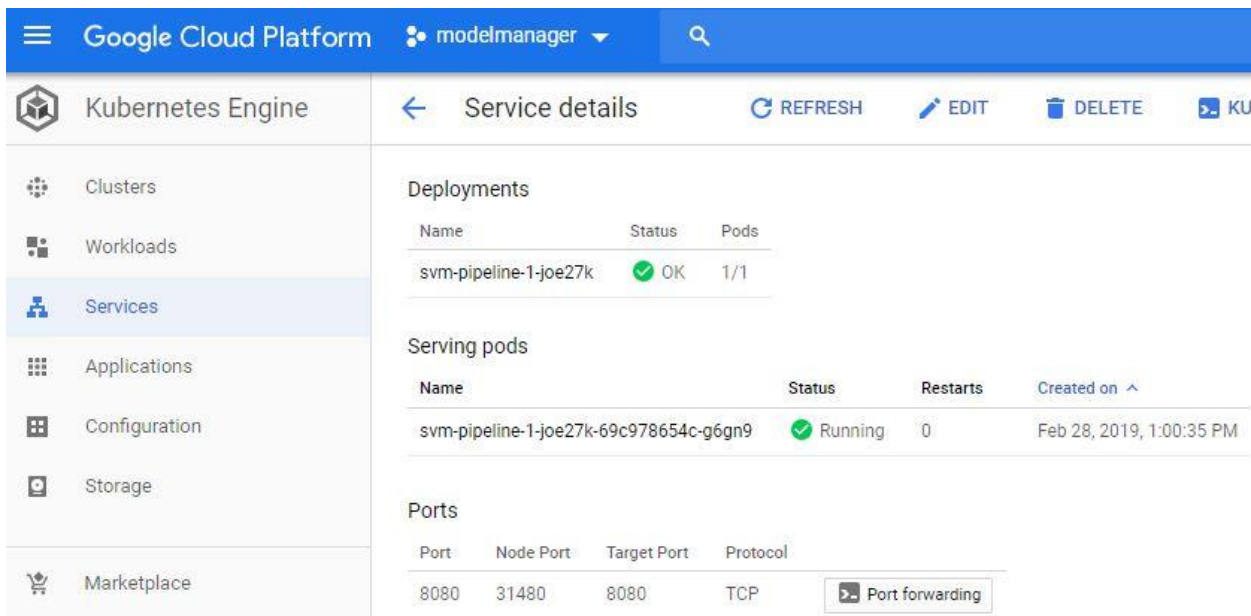


Figure 31. GCP – Verify Service Pod in Google Kubernetes Engine

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_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.

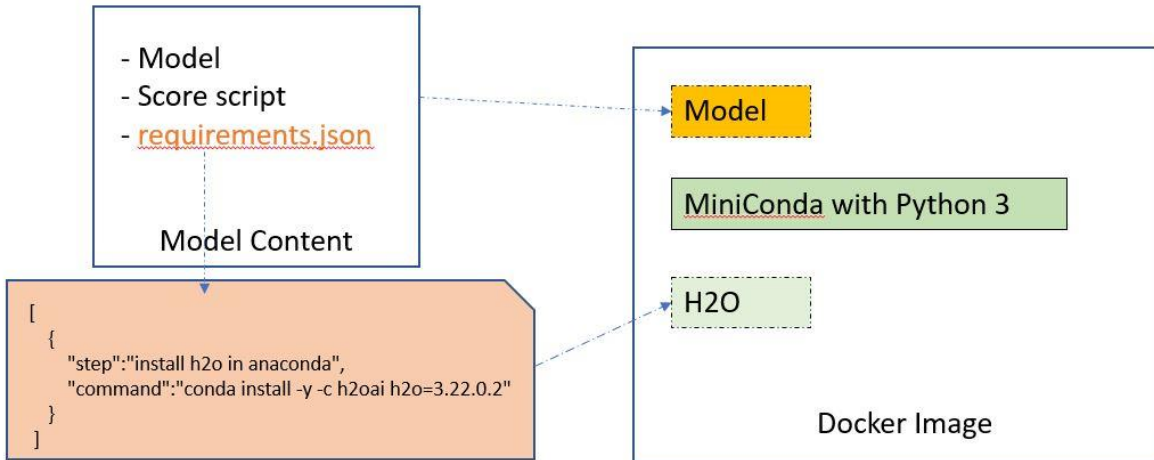


Figure 34. Support Extra Model Dependency

Figure 35 shows the specification file in the model content.

The screenshot shows a file editor window titled 'XGBoost (1.0)'. The left sidebar shows a file tree with 'requirements.json' selected. The main editor area displays the content of 'requirements.json' (Read-Only) as follows:

```

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-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-ac3cc9557c57:1.0'
```

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.

Paper SAS4536-2020

Choose Your Own Adventure:

Manage Model Development via a Python IDE

Jon Walker, SAS Institute Inc.

ABSTRACT

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

INTRODUCTION

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

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

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

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

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

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

MODEL MANAGEMENT

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

SWAT MODEL

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

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

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

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

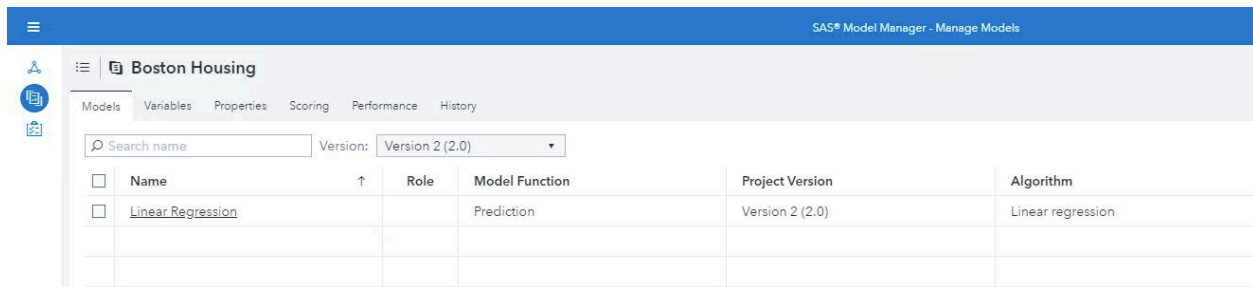
Everything in sasctl requires an established session to a SAS Viya server. At its most basic level, sasctl is a sophisticated REST client that calls the REST APIs available in all SAS Viya environments. Creating a session allows sasctl to repeatedly call those APIs on your behalf without requiring you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

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

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

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

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



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

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

Display 1. Model in SAS Model Manager

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

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

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

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

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

30.003843377

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

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

SCIKIT-LEARN MODEL

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

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

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

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

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

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

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

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

Display 2. Updated SAS Model Manager Project

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

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

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

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

```

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

```

Display 3. Files Uploaded to SAS Model Manager

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

SAS® Model Manager - Manage Models

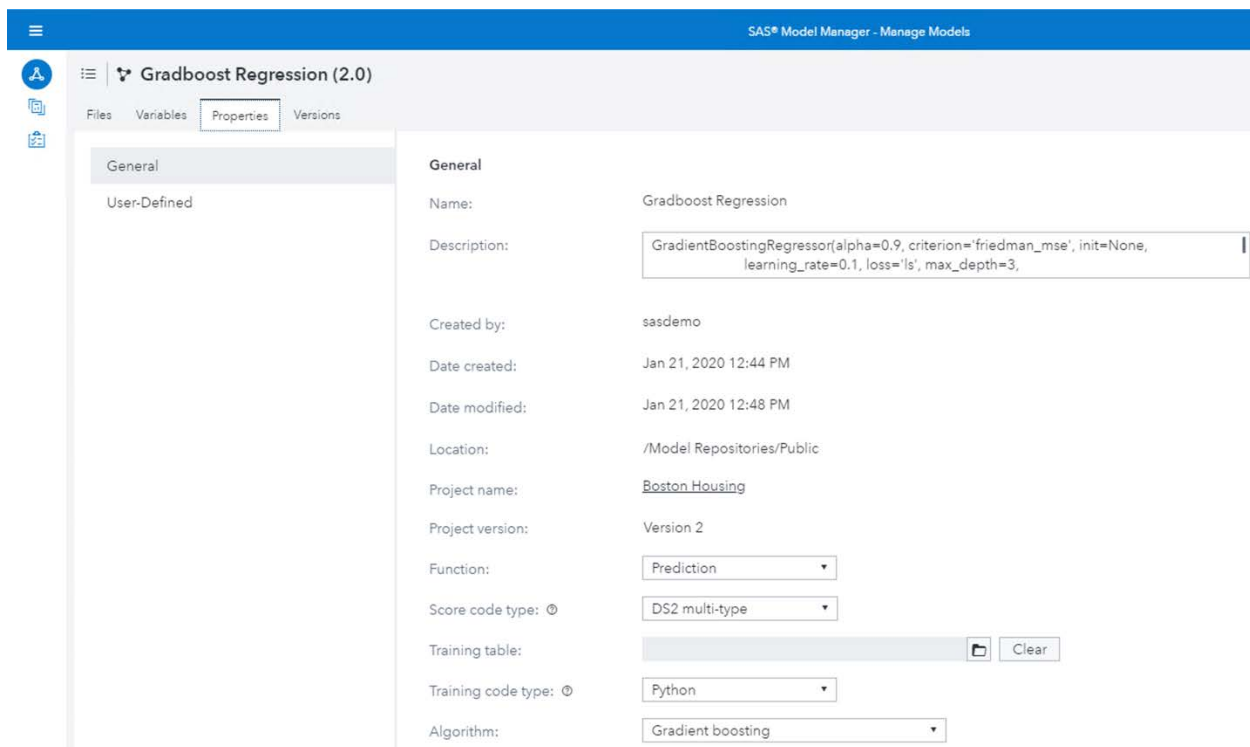
Gradboost Regression (2.0)

Files Variables Properties Versions

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

Display 4. Input and Output Variables in SAS Model Manager

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



Display 5. Model Properties in SAS Model Manager

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

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

```

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

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

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

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

```

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

```

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

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

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

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

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

SUPPORTING TASKS

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

PERFORMANCE MONITORING

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

```

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

```

```

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

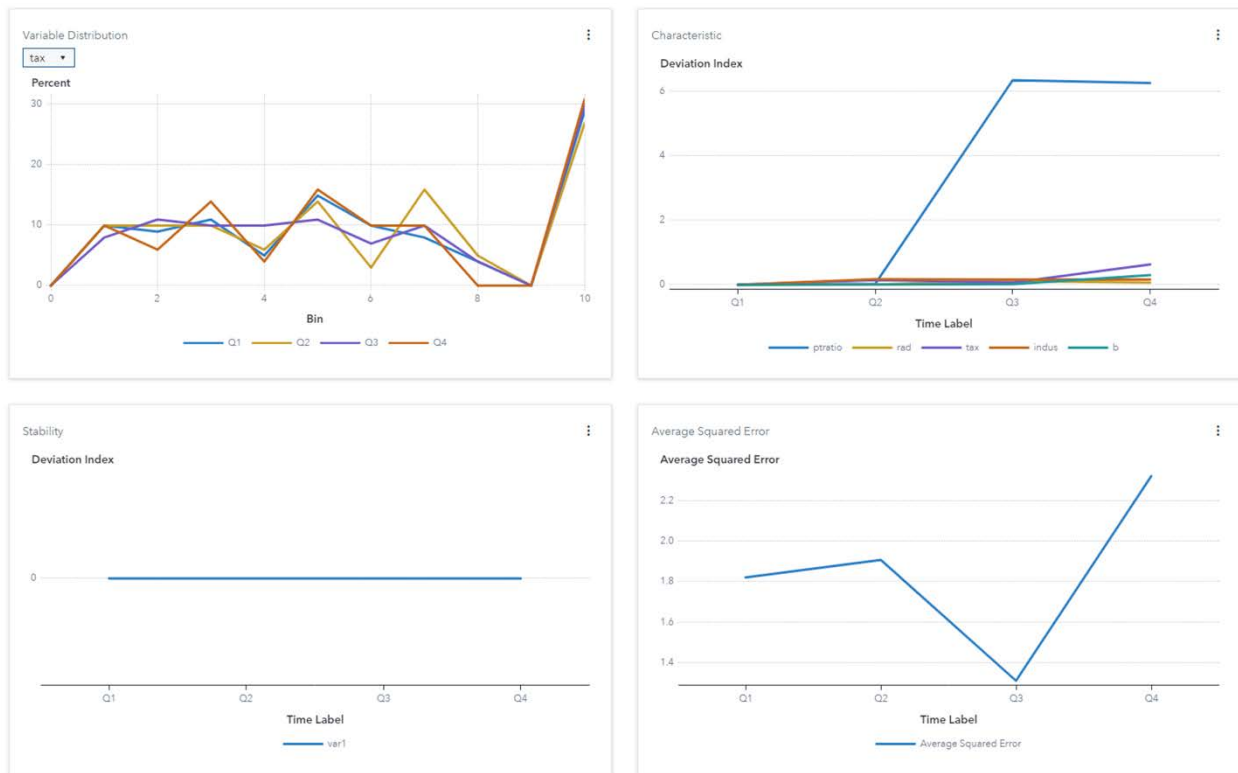
```

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

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

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

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



Display 6. Model Manager Performance Reports

REPORT VISUALIZATIONS

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

```

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

```

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

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

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

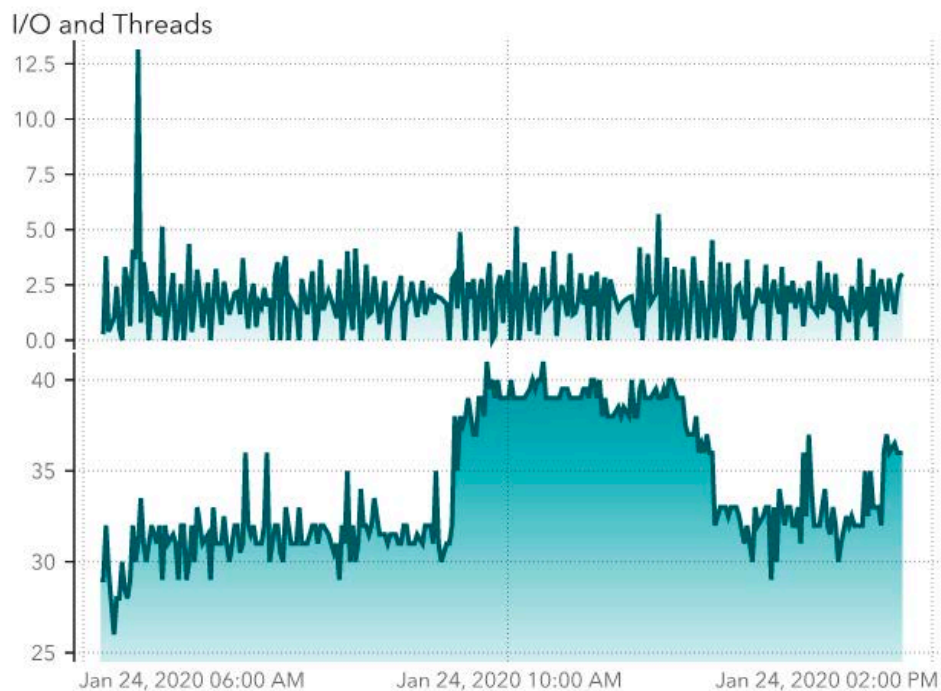


Figure 1. CAS Server Activity

LOW-LEVEL USAGE

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

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


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

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

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

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

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

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

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

Output 4. The “Files” Link

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

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

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

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

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

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

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

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

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

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

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

Output 6. Content of the traincode.sas File

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

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

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

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

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

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

CONCLUSION

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

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

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

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

APPENDIX

INSTALLING SASCTL

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

```
pip install sasctl
```

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

- `requests`
- `six`

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

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

ESTABLISHING SESSIONS

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

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

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

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

ACKNOWLEDGMENTS

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

CONTACT INFORMATION

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

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

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

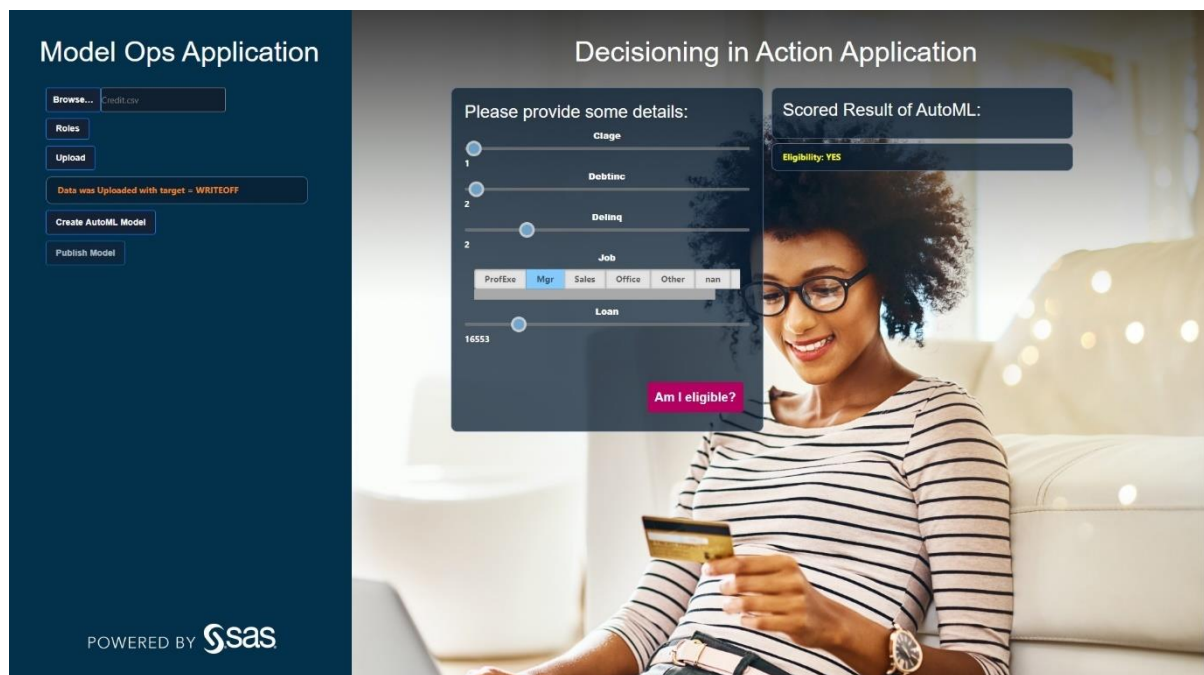
Other brand and product names are trademarks of their respective companies.

Build Your ML Web Application Using SAS AutoML

By Paata Ugrekhelidze

Published on The SAS Data Science Blog, May 15, 2020

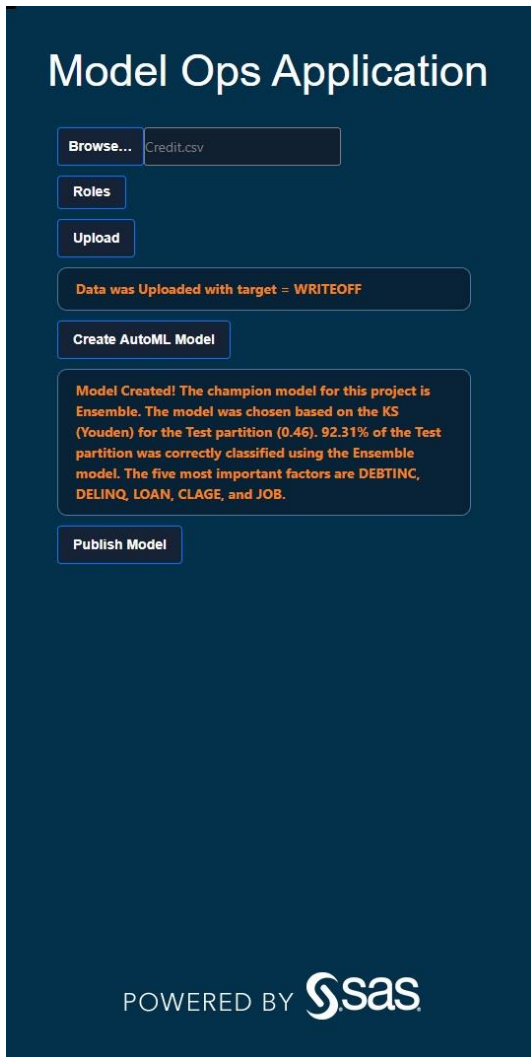
Most people require loans at some point: student loans, car loans, mortgage, etc. Using an online loan service, I scored a great deal on a vehicle. The moment I saw the car, I immediately applied for a loan. To my surprise, I was approved that same day, which allowed me to purchase it before other potential buyers. Online loan applications are automated processes that allow people to quickly receive loans without unnecessary delays. Nowadays, automated decision-making is dominated by machine learning algorithms. However, algorithms require a lot of manual and technical capabilities to be practical and effective. SAS AutoML offers the ability to automate the development of machine learning algorithms. This article will demonstrate the implementation of AutoML in banking loan applications.



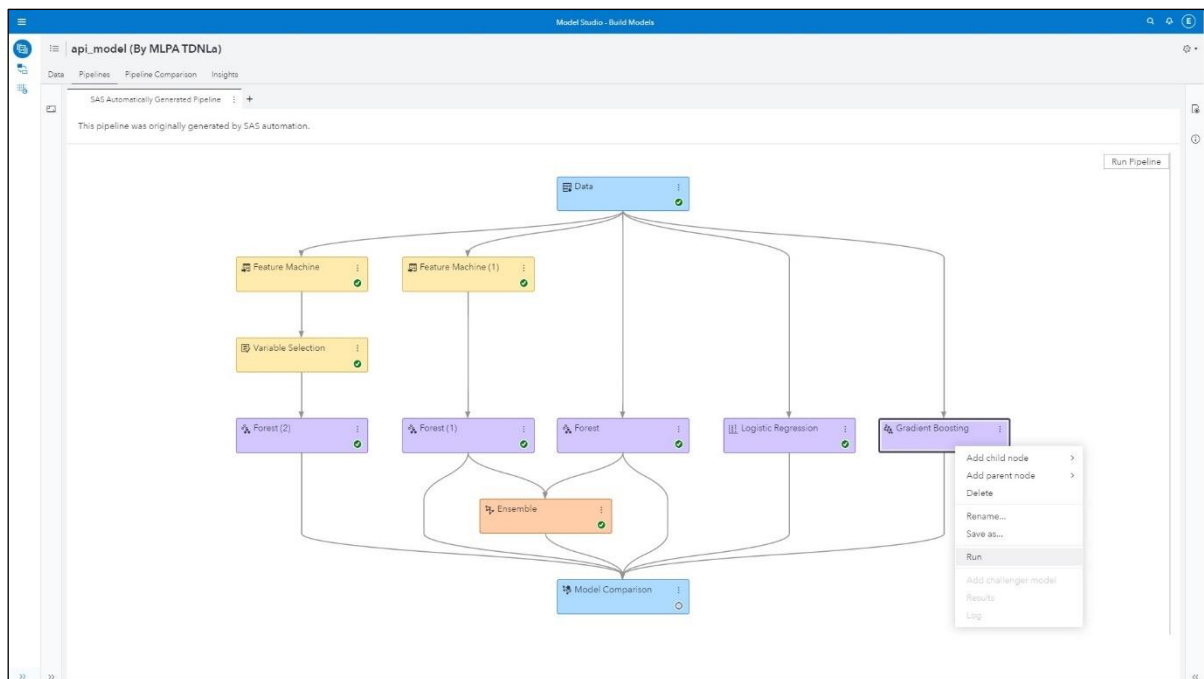
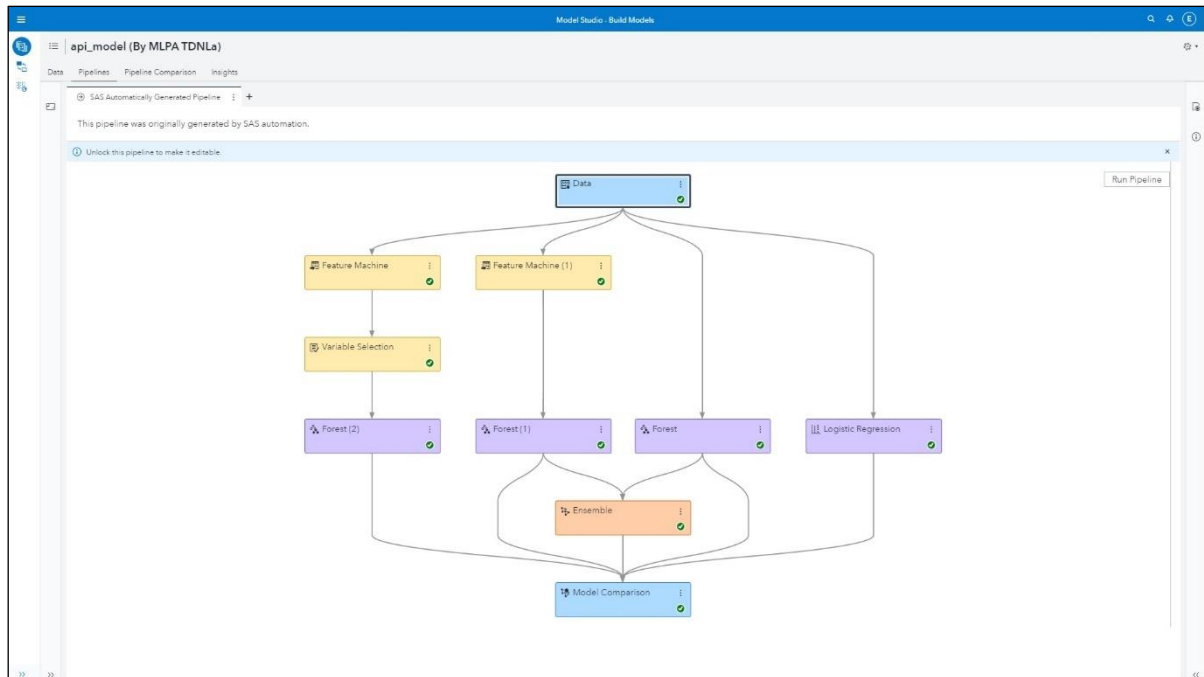
The image above shows two separate applications. *Decisioning in Action* (right) shows a customer who is trying to apply for a loan application. After filling out their information, one click on the *Am I eligible?* button can decide whether they are eligible to receive a loan.

2 SAS and Open-Source Model Management

Model Ops (left) shows the point-of-view of a data scientist who is tasked to create a model that will decide whether to approve a loan application. The *Create AutoML Model* button invokes an API to develop automated model.

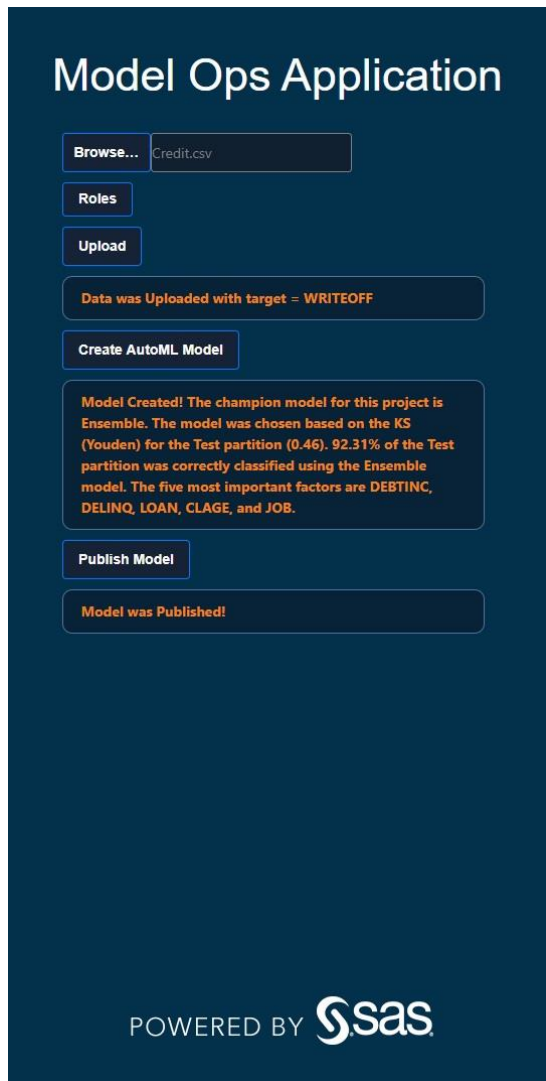


However, the model is not a “black box” in many ways. AutoML could be configured before creation, such as how long to train a model(s). In addition, algorithms created by AutoML can be viewed and customized.



The ability to view and change models makes AutoML less of a “black box” algorithm and more of a “model recommendation” for data scientists to refer to. Models can be governed, monitored, and deployed (in containers, cloud servers such as EC2 instances, and so on) using SAS Model Manager on SAS Viya by simply clicking on *Publish Model*.

Note: SAS Model Studio, the tool that developed this model, requires a SAS VDMML license. SAS Model Manager, where models are registered for governance, monitoring, and deployment, is available with a SAS MM license.

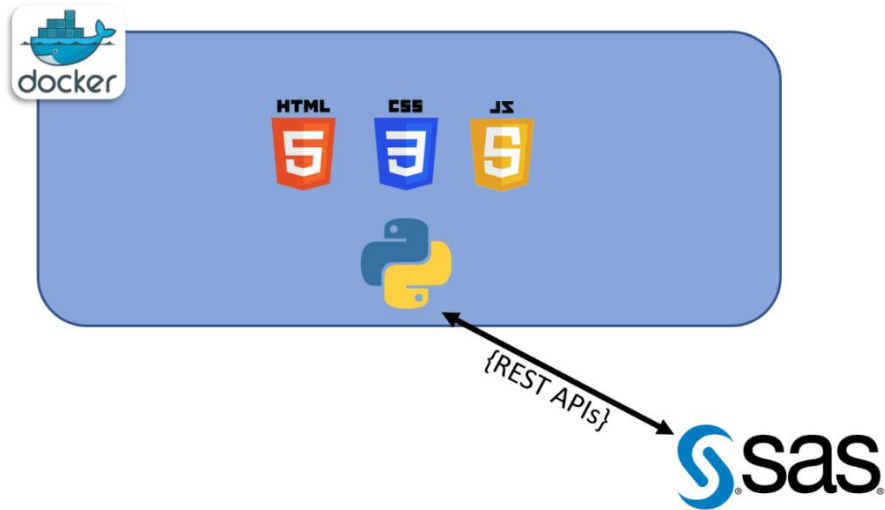


Conclusion

Developing a loan approval application is a sensitive task since automatically approving loans to customers who will default can be costly for a lender. SAS enables quick and easy development and exposure of decision-making models from a single source. A simple and robust environment can make decisions less prone to errors. In case you are wondering how the application was created, the section below generalizes the main components and how SAS interacts with the application.

Deployment (Bonus)

The diagram below describes the steps it took to develop the application. For the front-end side of the application, *HTML*, *CSS* and *Javascript* were used to populate, design, and make the web content dynamic. On the other hand, *Python* language was used for the back-end to run a lightweight web framework and transfer data between the front-end & *SAS* environment via *REST APIs*. Javascript could have been used for the back-end to call SAS functions as an alternative because API calls can be executed from many languages. The application was containerized and can be easily deployed from other machines with *docker* platform.



Monitoring the Relevance of Predictors for a Model Over Time

Ming-Long Lam, Ph.D., SAS Institute Inc.

ABSTRACT

This paper presents a novel approach to monitor model performance over time. Instead of monitoring accuracy of prediction or conformity of predictors' marginal distributions, this approach watches for changes in the joint distribution of the predictors. Mathematically, the model predicted outcome is a function of the predictors' values. Therefore, the predicted outcomes contain intricate information about the joint distribution of the predictors. This paper proposes a simple metric that is coined the Feature Contribution Index. Computing this index requires only the predicted target values and the predictors' observed values. Thus, we can assess the health of a model as soon as the scores are available and raise our readiness for preemptive actions long before the target values are eventually observed. This index is model neutral because it works for any types of models that contain categorical or continuous predictors, and models that generate predicted values or probabilities. Models can be monitored in near real time since the index is computed using simple and time-matured algorithms that can be run in parallel. Finally, it is possible to provide statistical control limits on the index. These limits help foretell whether a particular predictor is a plausible culprit in causing the deterioration of a model's performance over time.

INTRODUCTION

In today's intelligence-driven economy, corporations increasingly rely on analytic models to make their business decisions. Like all tangible assets, models become dated, and their accuracies diminish over time. To stay competitive, corporations constantly monitor their models. When signs of deterioration of model performance appear, stakeholders need to determine if the models must be proactively updated or rebuilt to correct the problems. Since every decision to refresh a model carries risks and can disrupt normal business, a solid business case must be presented to support the request to update or rebuild a model.

Not all models can be monitored or are worth monitoring. In this paper, we focus on monitoring supervised learning models where there is one target variable. Most, if not all, model performance metrics have one thing in common: they measure how well the model predicted values agree with the observed target values. Various model performance metrics have been developed to measure the degree of this agreement. However, we sometimes need to assess the health of a model at the time of scoring when the target values are yet to be observed. If we must wait for the availability of the observed target values, then we might lose the opportunity to make a time-sensitive decision to refresh the model sooner. An example of this need is the fraud detection model. It is known that those who commit fraud game the system to avoid being detected. Since it takes time to diligently investigate fraud, we need some indicators to tell us now if the current system is being gamed. If we find that the system is no longer effective in detecting fraud, then countermeasures must be taken to correct the situation.

Although there are currently various model performance metrics to measure the overall performance of a predictive model, not all metrics are able to pinpoint which predictors might be responsible for the deterioration of model performance. In addition, some metrics are applicable only to certain types of models and computing the metrics might require us to rebuild the model.

This paper proposes a model performance metric coined the Feature Contribution Index. Only the predicted target values and the **predictors' values are needed** to calculate the index. Thus, we can assess the health of a model as soon as the model scores are available and take preemptive actions long before the target values are observed. This index is model neutral because it works for any types of models that contain categorical or continuous predictors, and models that generate predicted values or predicted probabilities. Models can be monitored in near real time since the index is computed using simple and robust algorithms that can be run in parallel. Finally, statistical control limits on the index can be provided. The statistical control limits can help determine whether a particular predictor is **causing the deterioration of a model's** health over time.

IDEA CONCEPTION

Loosely speaking, a supervised learning model is an algorithm that takes the values of predictors as inputs and computes the predicted value of the target variable as the output. The algorithm is constructed based on the assumptions made on the probability distribution of the data. The assumptions are the relationship between the target variable and the predictors, the covariance structure of the predictors, and the distribution of the target variable. In a linear regression model, for example, the relationship between the target variable and the predictors is linear, the covariance structure of the predictors is fixed, and the target variable follows a normal distribution that is parametrized by a mean and a constant variance. The mean of the target variable is the predicted value of the linear relationship.

When a supervised learning model does not perform, we mean that the model can no longer be used to describe the probability distribution of the current data. In other words, some assumptions made are no longer valid. Different metrics have been developed to check specific assumptions. For the linear regression model, the lack-of-fit test checks the linear relationship assumption. The test of homogeneity checks the constant target variable variance assumption. The Shapiro–Wilk test checks the normality assumption. In summary, checking the relationship between the target variable and the predictors and determining the probability distribution of the data assumptions requires the observed target values. When the target variable has yet to be observed, which is a common situation in applying the model to new data, these two groups of assumptions cannot be checked. However, we can still check for changes in the covariance structure of the predictors.

To compare the covariance structure of the predictors over time, we can use multivariate tests of equality of covariance matrices such as **Box's M test**. **If we can put aside the** argument of whether these tests can apply to our data (due to the assumptions requiring observed target values), these tests can be helpful. However, we often need to know which of the predictors have triggered the differences in the covariance structures in addition to simply knowing that the covariance structures have changed over time.

Let us study the linear regression model to generate ideas. Under this model, the predicted value of the i -th observation is $\hat{y}_i = b_0 + \sum_{r=1}^k b_r x_{ir}$ where k is the number of predictors, b_0 is the intercept, b_1, \dots, b_k are the estimated regression coefficients, and x_{i1}, \dots, x_{ik} are the values of the predictors in the i -th observation. Using the fact that $\bar{y} = b_0 + \sum_{r=1}^k b_r \bar{x}_r$, it can be shown that

$$(\hat{y}_i - \bar{y})(x_{is} - \bar{x}_s) = \sum_{r=1}^k b_r (x_{ir} - \bar{x}_r)(x_{is} - \bar{x}_s) \text{ for } s = 1, \dots, k.$$

Suppose n is the number of observations, then we have

$$\frac{1}{n-1} \sum_{i=1}^n (\hat{y}_i - \bar{y})(x_{is} - \bar{x}_s) = \sum_{r=1}^k b_r \left(\frac{1}{n-1} \sum_{i=1}^n (x_{ir} - \bar{x}_r)(x_{is} - \bar{x}_s) \right).$$

The left-hand side of the equation is the observed covariance between the predicted values and the values of the s -th predictor. The right-hand side of the equation is a linear combination of the observed covariances between the values of each predictor and the s -th predictor.

When we apply this linear regression model to new data, the estimated regression coefficients are considered fixed. If the covariances among the predictors of the new data are the same as that of the training data, then the covariances between the predicted values and the values of the predictors of the new data should also be the same as that of the training data. The contraposition of this condition says that if the covariances between the predicted values and the values of the predictors of the new data are different from that of the training data, then the covariances between the predicted values and the values of the predictors of the new data should also be different from that of the training data. Therefore, if we compare the covariances between the predicted values and the values of the predictors with that of the training data, then we might be able to tell if the covariance structures of the predictors have changed.

EXTENSION TO CATEGORICAL VARIABLES

Next, we will attempt to extend the idea in the last section to categorical target variables or predictors. **Let's take on the categorical target variable first. If we directly apply the idea in the previous section to the predicted category of a categorical target variable, then we must choose some thresholds for the predicted probabilities of the target categories. Instead of running into the arguments of choosing the "right" thresholds, we will apply the above idea to the predicted probabilities. In other words, we will use the correlation between each of the predicted probabilities with each of the predictors as our metrics.**

We cannot break a categorical predictor into its individual levels. Under this constraint, we need to look outside of correlation for our metric. The Eta-Square statistic measures the association between an interval variable and a categorical variable in a general linear model. When the general linear model has only one categorical predictor, the Eta-Square value is equal to the **model's** R-Square value. The **model's** R-Square value is the square of the correlation value when the general linear model has only one interval predictor. Therefore, to use the same metric for both categorical predictors and interval predictors, we adopt the R-Square value as our metric and coin it the Feature Contribution Index.

FEATURE CONTRIBUTION INDEX

For a classification model where the target variable is categorical, the model outcome consists of the predicted probabilities. For a regression model where the target variable is continuous, the model outcome is the predicted value. In both types of models, the model outcome consists of one or more numeric values. We measure individual predictors' contribution to model performance by using the following procedure:

1. For each numeric value in the model outcome, perform the main effect analysis of variance on each individual predictor.
2. Measure the contribution of this predictor by the R-square statistic. For a categorical predictor, this is the full Eta-Squared statistic. For an interval predictor, this is the squared Pearson correlation coefficient.
3. For a categorical target, aggregate the contribution indices calculated in step two for each individual predicted probability. The aggregation method is discussed in the next section

The Feature Contribution Index is a numeric value between zero and one inclusively. A value of one indicates that the predictor solely determines the model outcomes. A value of zero indicates that the predictor has no bearing on the model outcomes.

AGGREGATION FOR A CATEGORICAL TARGET

We calculate the Feature Contribution Index for each predicted probability in step two of the steps listed in the previous section. We want to come up with a single index for a categorical target since examining a single index is always more preferred than studying several indices for insights. This single index is a weighted sum of the individual Feature Contribution Indices. Without loss of generality, we require positive weights whose sum is one. We can consider two choices for weights. The first and the non-informative choice sets the weights equal to the reciprocal of the number of predicted probabilities. For example, the weights are 0.5 for a binary target. Another choice sets the weights equal to the observed relative frequencies (that is, proportions) of the target categories.

For a binary target variable, the choice of weight does not matter. Let p be the predicted probability of the event. Then $1 - p$ is the predicted probability of the non-event. For a predictor x , $\text{CORR}(x, 1-p) = -\text{CORR}(x, p)$. Therefore, $(\text{CORR}(x, 1-p))^2 = (\text{CORR}(x, p))^2$. Both sides of the equation are the R-Square statistics of the Feature Contribution Indices for the non-event and the event respectively. Since they are equal, any weighted sum of them will result in the same results provided the weights are positive and sum to one.

STATISTICAL CONTROL LIMITS

Since our original goal is to monitor a model over time, we are more interested in studying the change or the trend of the R-Square statistics of each predictor than the R-square statistics themselves. In other words, we are more interested in the change of the R-Square statistics over time compared to a benchmark. An obvious choice for the benchmark is the R-Square statistics that are calculated on the training data.

Our hypothesis is that the **predictors' covariance structure in each monitoring data is** identical to that of the training data. Under this hypothesis, the R-Square statistics that are calculated on the monitoring data should be ideally the same as the R-Square statistics that are calculated on the training data. In practice, the R-Square statistics are different because of the usual random elements in observing the data. Our question is how much differences among the R-Square statistics can we tolerate before we drop the hypothesis? We address this question by constructing a confidence interval for the R-Square statistics at each time point. If we can agree that the R-square statistic is equal to the Eta-Square statistic for an interval predictor, then we can apply the interval inversion method (Kromrey and Bell 2010 and Steiger 2004) to construct a confidence interval for the Eta-Square statistic.

Let η_0^2 be the Eta-Square statistic (that is, the benchmark value) calculated on the training data that have n observations. The corresponding F value is

$$F_0 = \eta_0^2 / (1 - \eta_0^2) \times df_2 / df_1 \text{ with two degrees of freedom, } df_1 \text{ and } df_2 = n - 1 - df_1.$$

For an interval predictor, $df_1 = 1$. For a categorical predictor, df_1 is equal to the number of categories of the predictor. Suppose the confidence interval will have $100p\%$ of coverage confidence, then, according to Kromrey and Bell (2010), the interval inversion method finds two values of the non-centrality parameter. One value is such that the observed F significance equals $(1 - p)/2$. Another value is such that the observed F significance equals $(1 + p)/2$. The FNONCT function in SAS[®] can calculate these two values of the non-centrality parameter for the F distribution. The function takes four arguments in the following order: the observed F value, the df_1 value, the df_2 value, and the desired F significance value. Since the FNONCT function uses a Newton-type algorithm to iteratively calculate a nonnegative non-centrality value, the function might return a missing value when the algorithm fails to converge. This might happen when the observed F value is relatively small.

Let $NCP_ETA_L = FNONCT(F_0, df_1, df_2, (1 + \rho)/2)$ and $NCP_ETA_U = FNONCT(F_0, df_1, df_2, (1 - \rho)/2)$. In a one-way analysis of variance, the non-centrality (NC) parameter of the F test is equal to the sum of squares of the model (SSM) divided by the mean squares error (MSE) ($MSE = SSE / df_2$). The Eta-Square statistic is equal to $SSM / (SSM + SSE)$. Thus, the Eta-Square statistic is equal to $NC / (NC + df_2)$. Using this simple relationship, the lower confidence limit for Eta-Square is $NCP_ETA_L / (NCP_ETA_L + df_2)$ and the upper confidence limit for Eta-Square is $NCP_ETA_U / (NCP_ETA_U + df_2)$. Please beware that the formula that we used is slightly different from that in Kromrey and Bell (2010). In their paper, Kromrey and Bell used the sample size n instead of df_2 . Because of this difference, our control limits are slightly wider than the control limits proposed by Kromrey and Bell.

When we score the model on a monitoring data set, **we assume that the predictors' covariance structure remains unchanged** from that of the training data. Under this assumption, we would expect the *actual* Eta-Square values of the monitoring data are not different from the benchmark values. Therefore, we will use the η_0^2 for constructing the limits. The df_1 value stays the same despite the possibility of not observing all the categories of the categorical predictor (otherwise, we unknowingly changed the covariance structure). The df_2 value is equal to the number of observations in the monitoring data. Since we treat η_0^2 , which is itself a random variable, as fixed benchmarks for the monitoring data, we should call the limits the control limits to avoid any statistical issues. This is because we cannot ensure that the coverage confidence is actually the nominal value 100%.
100%.

Finally, if the Eta-Square statistics that are calculated on the monitoring data are outside the control **limits**, **then we have reasons to suspect that the predictors' covariance structure** might have changed.

SAS MACROS

The following three SAS macros were developed for calculating the Feature Contribution Index:

1. The `Compute_FCI_NomPred` macro computes the Feature Contribution Indices for a list of categorical predictors.
2. The `Compute_FCI_IntPred` macro computes the Feature Contribution Indices for a list of interval predictors.
3. The `Compute_FCI` macro reads input specifications, calls the `Compute_FCI_NomPred` and the `Compute_FCI_IntPred` macros to compute the Feature Contribution Indices, and returns the indices in the specified data.

These macros require that the model outcomes are already available in the input monitoring data. You can download these macros from <https://support.sas.com/downloads/package.htm?pid=2225> and the accompanying documentation from http://support.sas.com/documentation/prod-p/mdlmgr/14.2/en/PDF/SMM142_FCI_Macros.pdf.

A fourth macro, `Create_FCI_Report`, was later developed for facilitating the entire process. It bypasses the `Compute_FCI` macro and directly calls the `Compute_FCI_NomPred` macro and the `Compute_FCI_IntPred` macro. In addition, it generates professionally formatted tables and charts. In the future, the `Create_FCI_Report` macro will be available for download and publish. In the meantime, interested readers can contact the author directly to obtain the `Create_FCI_Report` macro.

SIMULATION STUDY

After we **determine that the predictors' covariance structure has changed over time**, our next task is to determine which of the predictors have triggered the change. This task is more difficult than it looks.

Let us use the linear regression example to aid our discussion. Recall that the covariance between a predictor and the predicted value is a linear combination of the individual covariances between two predictors. Mathematically, this is

$$\frac{1}{n-1} \sum_{i=1}^n (\hat{y}_i - \bar{y})(x_{is} - \bar{x}_s) = \sum_{r=1}^k b_r \left(\frac{1}{n-1} \sum_{i=1}^n (x_{ir} - \bar{x}_r)(x_{is} - \bar{x}_s) \right).$$

Without loss of generality, let us consider the scenario where only the covariance between the first two predictors has changed but not their individual variances. This change will affect the covariance between the first predictor and the predicted value. Convoluted with the sign of the regression coefficient of the first predictor in the benchmark model, the covariance between the first predictor and the predicted value will increase or decrease. Similarly, the covariance between the second predictor and the predicted value will increase or decrease. When we notice that only the Feature Contribution Indices of a pair of predictors are outside the control limits, we might conclude that the covariance between that pair of predictors has changed.

In another scenario, the variance of the first predictor has changed. In other words, the distance between $x_{i1} - \bar{x}_1$ has changed for $i = 1, \dots, n$. This change will affect all the covariances that involve the first predictor. Since either $r = 1$ or $s = 1$, all the covariances between a predictor and the predicted value will be affected. When we notice that all the Feature Contribution Indices are outside the control limits, we need to compare the **predictors' variance** and their covariances with the benchmark value. An inadequacy of the Feature Contribution Index is the scenario where we do not come to any conclusions after studying these comparisons.

In our final scenario, only the mean of the first predictor has changed. This will not affect any covariances. Thus, we will not see any Feature Contribution Indices outside the control limits. If we are not concerned about the shifts of any means, then this is a good feature of the Feature Contribution Index as we have one thing less to check. Otherwise, this is another inadequacy of the Feature Contribution Index and we must turn to other diagnostic methods instead.

We are going to use a simulation study to illustrate the above three scenarios, demonstrate the steps of calling the macros, and review the results. The predictors are, namely, X1 and X2. The target variable is y whose expectation is $E(Y) = -3 + 4 * X1 + 2 * X2$.

A normal random noise with zero mean and unit variance is added to $E(Y)$ to obtain the observed Y. In the training data, which contains 1000 observations, the predictors have zero means, unit variances, and are uncorrelated (that is, zero correlations). The ordinary least squares estimate of the model is $\hat{y} = -2.9762 + 4.0061 * X1 + 1.9721 * X2$.

SCENARIO 1: CHANGE CORRELATION

Seven monitoring data sets, each containing 100 observations, are simulated. The **predictors' covariance structures are the same as that in the training data except for the correlation between the two predictors**. Here are the changes to the correlations that are simulated in the five monitoring data sets:

1. The correlation between X1 and X2 is 0.0, i.e., no change.
2. The correlation between X1 and X2 is -0.4.

3. The correlation between X1 and X2 is -0.2.
4. The correlation between X1 and X2 is +0.2.
5. The correlation between X1 and X2 is +0.4.

After we score the data sets, our first instinct is to compare the distributions of the predicted values with that of the benchmark training data (the ID column). The box-plots in Figure 1 help us visualize the comparison. At a first glance, although the distributions have different ranges, they are visually indifferent based on metrics such as the means and the medians. If we compare their interquartile ranges, then we may suspect that the distributions in the second and the fifth monitoring data sets have changed because their interquartile ranges are shorter than others. Therefore, visually comparing distributions might not enable us to realize **that the predictors' covariance structures have changed**.

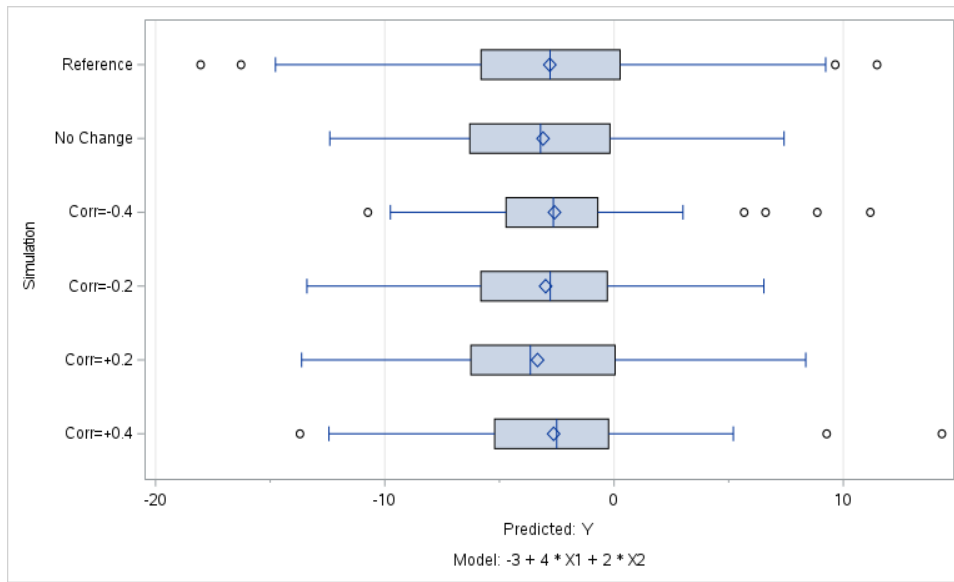


Figure 1. Box Plots of Predicted Values Across Monitoring Data Sets for Scenario 1

The panel chart in Figure 2 shows the Feature Contribution Indices of the predictors across the monitoring data sets. The benchmark index is subtracted from the indices and the control limits so that all the graphs are drawn using the same scale. Therefore, the vertical axis tells the change from the benchmark index. Finally, the graphs are shown in descending level of the benchmark indices (the value of the level is inside the parentheses of the individual chart titles). This enables us to focus on the predictors that contribute more to the model outcome in the benchmark data.

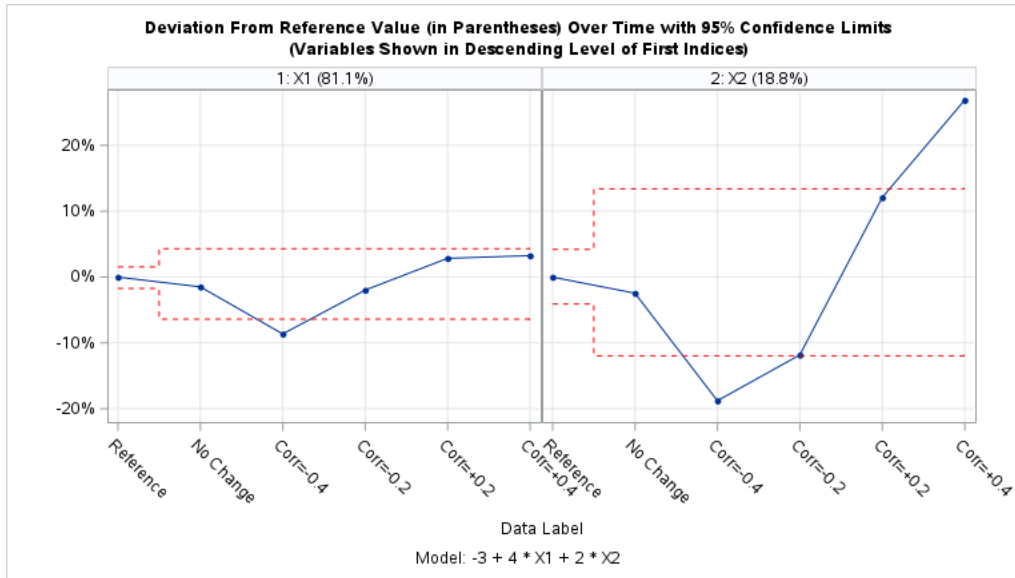


Figure 2. Feature Contribution Indices of Predictors Across Monitoring Data Sets for Scenario 1

Table 1 shows the Feature Contribution Indices of the five monitoring data sets and the benchmark training data (the Reference column). The out-of-bound values are highlighted in red.

Table 1. Feature Contribution Indices of Predictors for Scenario 1

Predictor	Reference	No Change	Corr=-0.4	Corr=-0.2	Corr=0.2	Corr=0.4
X1	81.1%	79.6%	72.5%	79.1%	84.0%	84.4%
X2	18.8%	16.3%	0.0%	7.9%	30.9%	45.6%

Ideally, we want to see that the indices of both perturbed predictors in each monitoring data set are outside the control limit (for example, X1 and X2 in the fourth monitoring data set that is labeled Corr=0.2). Since the monitoring data has only 100 observations, which is one-tenth the number of observations in the training data, the ideal results may not occur in every scenario unless the correlations become apparently stronger.

SCENARIO 2: CHANGE STANDARD DEVIATION

Four monitoring data sets, each containing 100 observations, are simulated. The predictors' covariance structures are the same as that in the training data except for the standard deviations of the predictors. Here are the changes to the standard deviation that are simulated in the four monitoring data sets:

1. The standard deviations of X1 and X2 are 1, i.e., No Change.
2. The standard deviation of X1 is 0.8 and that of X2 is 1.
3. The standard deviation of X1 is 1 and that of X2 is 1.25.
4. The standard deviation of X1 is 0.8 and that of X2 is 1.25.

A comparison of the distributions shows more distinct differences than what was found in the first scenario (Figure 3). The ranges and the interquartile ranges are visually different. However, the medians are seemingly the same.

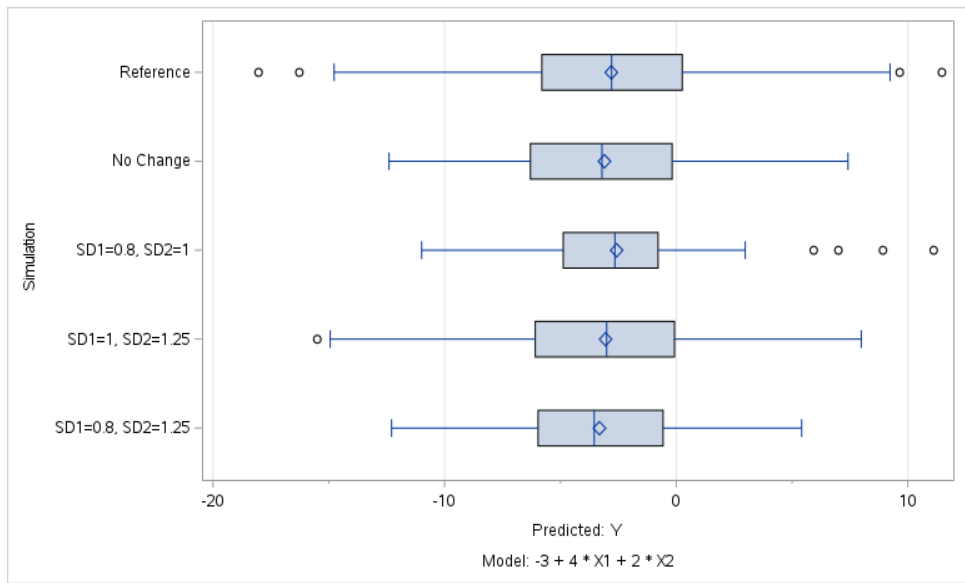


Figure 3. Box Plots of Predicted Values Across Monitoring Data Sets for Scenario 2

The panel chart in Figure 4 shows the Feature Contribution Indices of the predictors across the four monitoring data sets. Since X1 has the highest benchmark index, a change in the standard deviation of any predictor (including X1) in the monitoring data set will trigger a stronger ripple effect on the X1’s index. On the contrary, X2 has the lowest benchmark index, only a change in its own standard deviation plus another change of X1’s standard deviation in the monitoring data set can affect its index.

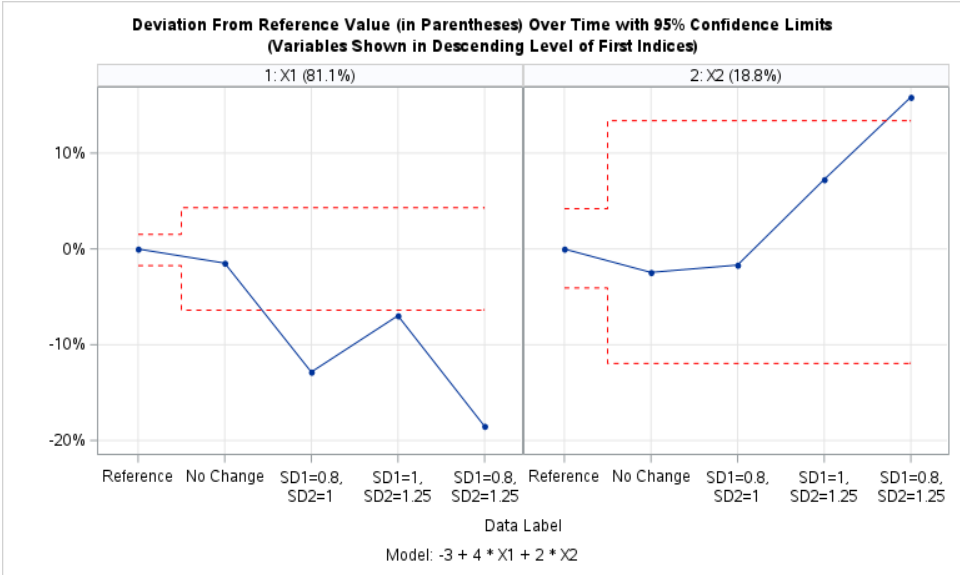


Figure 4. Feature Contribution Indices of Predictors Across Monitoring Data Sets for Scenario 2

Table 2 shows the Feature Contribution Indices of the four monitoring data sets and the benchmark training data (the Reference column). The out-of-bound values are highlighted in red.

Table 2. Feature Contribution Indices of Predictors for Scenario 2

Predictor	Reference	No Change	SD1 = 0.8 & SD2 = 1.0	SD1 = 1.0 & SD2 = 1.25	SD1 = 0.8 & SD2 = 1.25
X1	81.1%	79.6%	68.3%	74.1%	62.6%
X2	18.8%	16.3%	17.1%	26.0%	34.6%

Although we have not proved this fact mathematically, we notice that changing the standard deviation of a predictor will drastically magnify or shrink its Feature Contribution Index. For example, changing only the standard deviation of X2 from 1 to 1.25 in the third monitoring data set will magnify its index approximately 1.4 times (from 18.8% to 26.0%). The magnitudes of the changes might be even bigger when the standard deviations of other predictors also change.

SCENARIO 3: CHANGE THE MEAN

Four monitoring data sets, each containing 100 observations, are simulated. The predictors’ covariance structures are the same as that in the training data except for the means of the predictors. Here are the simulated changes to the mean in the four monitoring data sets:

1. The means of X1 and X2 are both 0, i.e., No Change.
2. The mean of X1 is -10 and that of X2 is 0.
3. The mean of X1 is 0 and that of X2 is 20.
4. The mean of X1 is -10 and that of X2 is 20.

The box-plots in Figure 5 show that the distributions have very different medians, but the ranges are similar. This is expected since we changed only the means of the predictors but not their covariance structures.

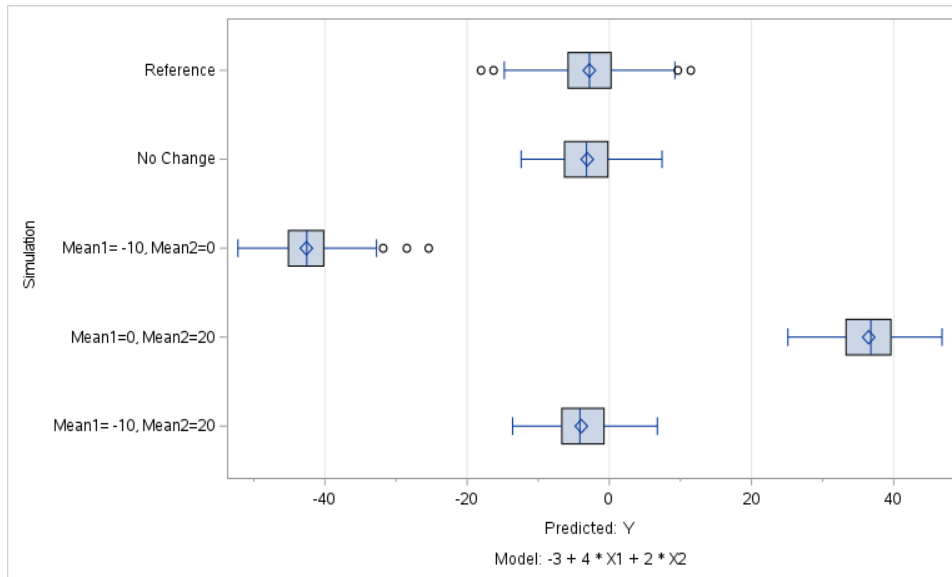


Figure 5. Box Plots of Predicted Values Across Monitoring Data Sets for Scenario 3

The panel chart in Figure 6 shows the indices of the predictors across the data sets. All the indices are within the control limits. We do not expect this because the indices are designed to detect changes in the covariance structures (including standard deviations and correlations), but not changes in the means.

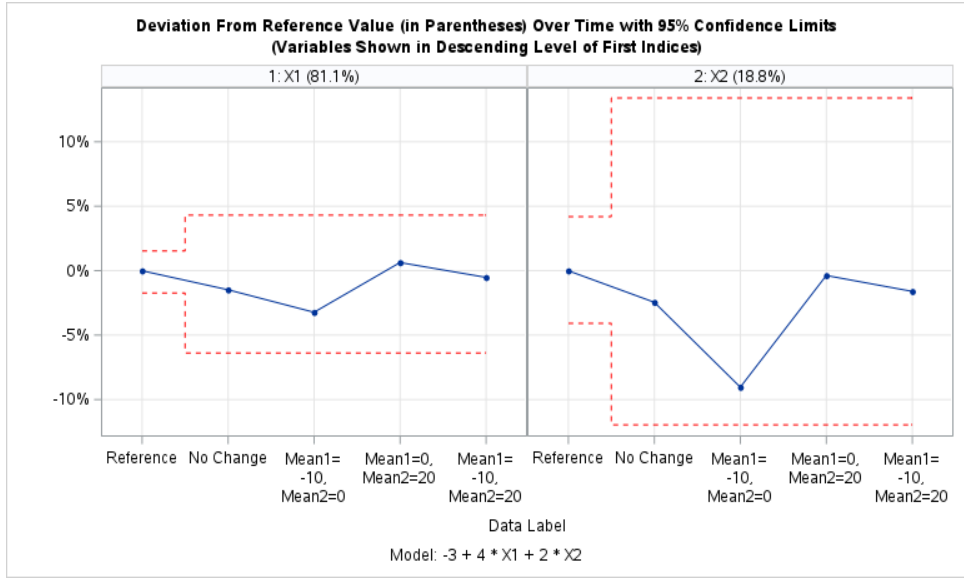


Figure 6. Feature Contribution Indices of Predictors Across Monitoring Data Sets for Scenario 3

Table 3 shows the Feature Contribution Indices of the four monitoring data sets and the benchmark training data (the Reference column). The out-of-bound values, if any, are highlighted in red.

Table 3. Feature Contribution Indices of Predictors for Scenario 3

Predictor	Reference	No Change	Mean1 = -10 & Means = 0	Mean1 = 0 & Means = 20	Mean1 = -10 & Means = 20
X1	81.1%	79.6%	77.9%	81.8%	80.6%
X2	18.8%	16.3%	9.7%	18.4%	17.2%

ANALYSIS EXAMPLE

Finally, we will illustrate our method using a real-life data set. The data in this example describes the historical usage patterns along with the weather data about the bike rental demand in the Capital Bikeshare program in Washington, D.C. The data is available on the Kaggle site¹. The original data was provided by Fanaee-T and Gama (2014).

For the sake of discussion, we are going to build a Poisson regression model to predict the total number of rentals (that is, count). The data originally covers 10,886 rental records dated from January 1, 2011 to December 31, 2012. We create the training data and four monitoring data sets based on the rental dates. The training data consists of all rentals in 2011 and it has 5,422 observations. The first monitoring data set consists of rentals in the first quarter of 2012 (that is, January to March) and has 1,363 observations. The second monitoring data set consists of rentals in the second quarter of 2012 (that is, April to June) and has 1,366 observations. The third monitoring data set consists of rentals from the third quarter of 2012 and it has 1,368 observations. Finally, the fourth monitoring data set consists of rentals from the fourth quarter of 2012 and has 1,367 observations.

A few categorical variables are created so that the Poisson regression model is more predictive. For example, the rental_hour_group variable is created by grouping values of the rental_hour variable. The grouping is determined mostly due to business reason. The

¹ <https://www.kaggle.com/c/bike-sharing-demand>.

Poisson regression algorithm converged. Plotting the predicted counts versus the observed counts assure us that the model fits the data well (Figure 7). Thus, we will use this model result for our discussion.

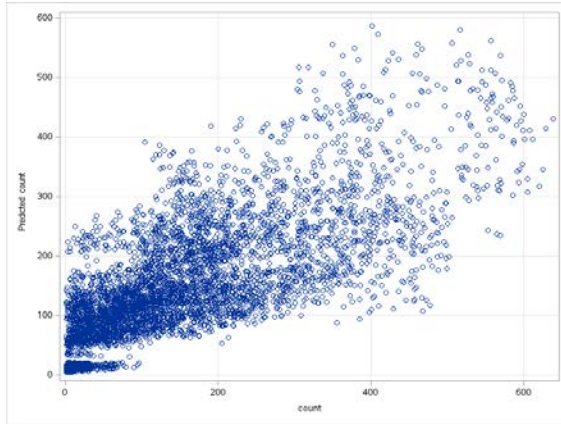


Figure 7. Predicted Counts Versus the Observed Counts of the Poisson Regression Model

Table 4 shows the parameter estimates of the Poisson regression model. The type III likelihood ratio tests of all the predictors are significant at the 0.01% level.

Table 4. Parameter Estimates of the Poisson Regression Model

Measurement Level	Parameter	Level	DF	Estimate
	Intercept		1	3.9253
Nominal	holiday	0	0	0
		1	1	0.0539
	rental_weekday	1	1	-0.0270
		2	1	-0.0665
		3	1	-0.0562
		4	1	-0.1220
		5	1	-0.0518
		6	1	0.0144
		7	0	0
	rental_hour_group	2AM - 5AM	1	-1.9459
		6AM - 8AM	1	0.6220
		9AM - 11AM	1	0.4495
		12NOON - 4PM	1	0.6151
		5PM - 7PM	1	1.0955
		8PM - 1AM	0	0
weather	1	0	0	
	2	1	-0.0602	
	3	1	-0.4314	
Interval	temp		1	0.0406
	humidity		1	-0.0014
	windspeed		1	-0.0037

Next, we will apply this model to the four monitoring data sets. Figure 8 shows the Feature Contribution Indices of all predictors in the training data and the four monitoring data sets. Overall, there are no drastic changes among the indices. The more noticeable changes are at the humidity (the relative humidity), the season (the season indicator), and the temp (the hourly temperature) predictors. Since the monitoring data sets are characterized by the rental dates (for example, in the first monitoring data set, season equals 1 for all observations, and humidity and temp varies within the winter-characterized ranges), the spreads of the humidity, the season, and the temp predictors in a monitoring data might be narrower than that in the benchmark data.

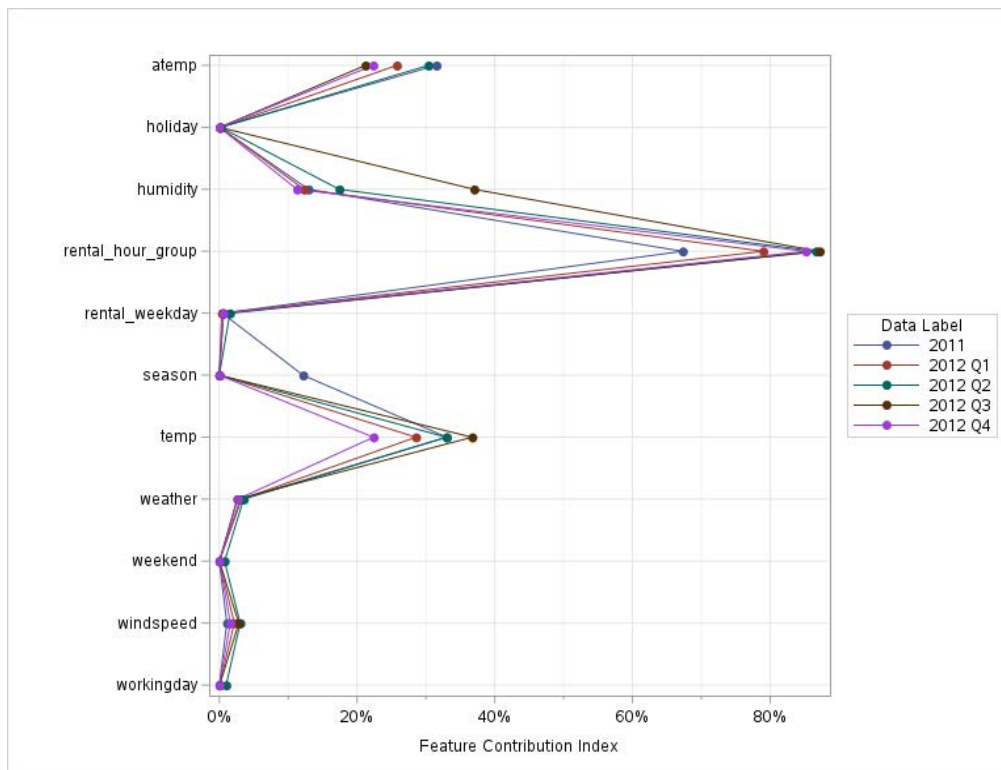


Figure 8. Feature Contribution Indices of the Monitoring Data for the Analysis Example

We will next review the Feature Contribution Indices of the predictors individually (Figure 9). You should notice that the weekend predictor does not have control limits. The LOG messages show that the FNONCT functions ran into computational problems and could not return values. Since the Feature Contribution Index of the weekend predictor is almost zero in the benchmark training data, we do anticipate this problem. Therefore, we can safely put this undesirable result aside.

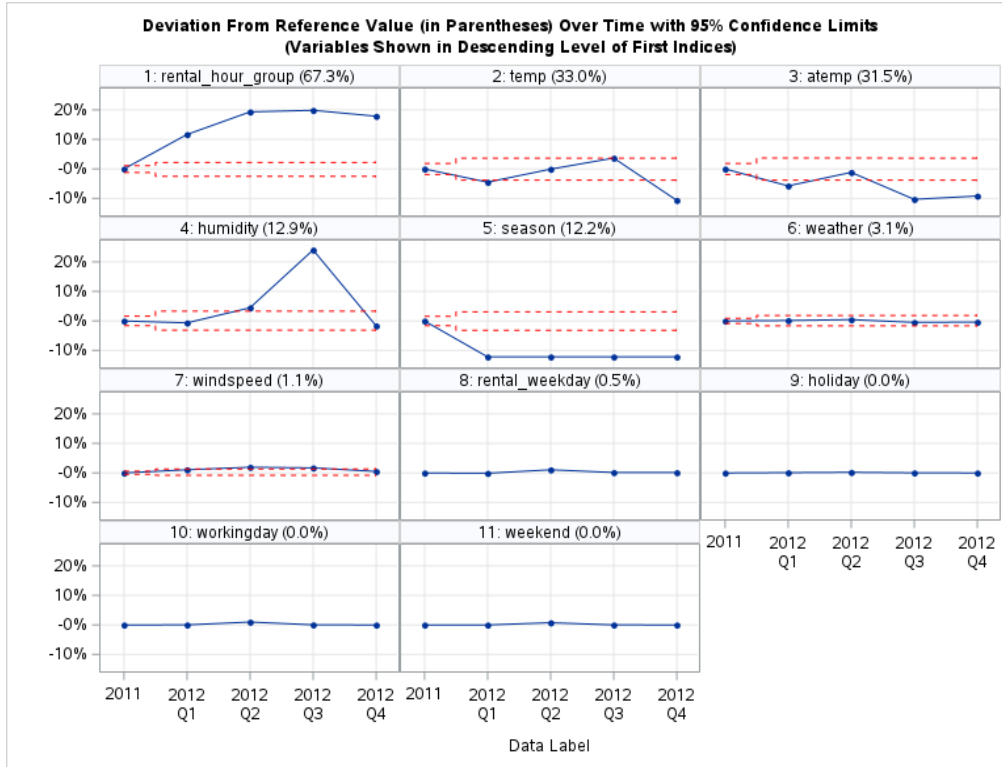


Figure 9. Feature Contribution Indices of Predictors Across Monitoring Data Sets for the Analysis Example

Table 5 shows the Feature Contribution Indices of the four monitoring data sets and the benchmark training data (the ID column). The out-of-bound values are highlighted in red.

Table 5. Feature Contribution Indices of the Predictors for the Analysis Example

Measurement Level	Predictor	2011	2012 Q1	2012 Q2	2012 Q3	2012 Q4
Interval	atemp	31.5%	25.8%	30.4%	21.2%	22.3%
	humidity	12.9%	12.3%	17.4%	37.0%	11.3%
	temp	33.0%	28.6%	33.0%	36.7%	22.4%
	windspeed	1.1%	2.2%	3.0%	2.8%	1.6%
Nominal	holiday	0.0%	0.1%	0.2%	0.1%	0.0%
	rental_hour_group	67.3%	79.0%	86.7%	87.2%	85.2%
	rental_weekday	0.5%	0.4%	1.5%	0.6%	0.6%
	season	12.2%	0.0%	0.0%	0.0%	0.0%
	weather	3.1%	3.2%	3.5%	2.6%	2.7%
	weekend	0.0%	0.0%	0.8%	0.0%	0.0%
	workingday	0.0%	0.0%	1.0%	0.1%	0.0%

The most obvious change occurs in the rental_hour_group predictor. The indices of other interval predictors that describe the climate of the quarters showed some changes. Our common sense tells us that these interval predictors are correlated, and their covariance structures do depend on the quarters. Figure 10 shows the association structure between the rental_hour_group and the humidity predictors by quarter.

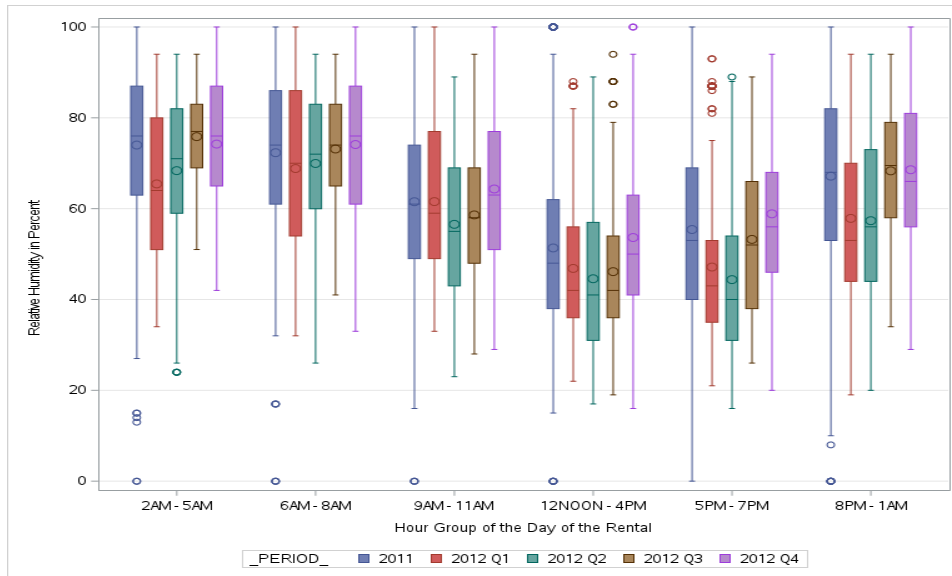


Figure 10. Association Structure Between the rental_hour_group and the humidity Predictors by Quarter

At first glance, the humidity predictor has a smaller range in the quarterly data than in the training data. In addition, the rental hour group seems to affect the interquartile ranges. For example, the interquartile ranges during 5PM – 7PM of 2012 Q1 overlaps the least with that of 2012 Q4. During other times of day, the interquartile range of 2012 Q1 overlaps more with that of 2012 Q4.

Finally, the Feature Contribution Indices of the season drop to zero in the four monitoring data sets. This is no surprise because the season predictor is practically constant in each monitoring data set. Thus, it has no relevance to the model outcome except to raise or lower the overall mean of the predicted rental counts.

CONCLUSION

We have introduced the Feature Contribution Index and we attempted to interpret the meanings of the index using a few simulated studies and the Bike Share Demand data. The Feature Contribution Index idea has plenty of room for improvement as we are not fully able to make conclusions based on their values. We welcome others to join us in further studying, improving, and interpreting the Feature Contribution Index.

REFERENCES

- Fanaee-T, H., and Gama, J. (2014). "Event labeling combining ensemble detectors and background knowledge", *Progress in Artificial Intelligence*, 2(2-3), 113-117, Berlin Heidelberg, Germany: Springer-Verlag. <https://doi.org/10.1007/s13748-013-0040-3>.
- Kromrey, J. D., and Bell, B. A. (2010). "ES_ANOVA: A SAS Macro for Computing Point and Interval Estimates of Effect Sizes Associated with Analysis of Variance Models," *Proceedings of the SouthEast SAS Users Group Conference (SESUG 2010)*, paper PO-05. Cary, NC: SAS Institute Inc. <https://analytics.ncsu.edu/sesug/2010/PO05.Kromrey.pdf>.
- Steiger, J. H. (2004). "Beyond the *F* Test: Effect Size Confidence Intervals and Tests of Close Fit in the Analysis of Variance and Contrast Analysis," *Psychological Methods*, 9(2), 164-182. <http://dx.doi.org/10.1037/1082-989X.9.2.164>.

CONTACT INFORMATION

Your comments and questions are welcomed, encouraged, and valued. In addition, you can request the SAS codes that generate the above results. Please contact the author at:

Ming-Long Lam
SAS Institute Inc.
(312) 995-5865
ming-long.lam@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.

Model Validation

A structured approach to model validation

By Hans-Joachim Edert

Published on The SAS Data Science Blog, February 19, 2020

This blog post is part one of a [series on model validation](#). The series is co-authored with my colleague [Tamara Fischer](#).

Finding the best approach for deploying analytical models into production is definitely not a new challenge. This final step of deployment – or “the last mile” as my colleague James called it in his [blog post](#) – has proven to be the most challenging part of the journey for many data scientists, and is often a stumbling stone.

One possible solution to this challenge is to apply the ideas of continuous integration / continuous delivery (CI/CD) to the area of analytical modeling. Since analytical decisions have become more crucial to business processes and deployment continues to be a challenge, it's worth exploring the benefits of these concepts.

A modelops deployment example

Recently, we were asked by a customer to showcase the benefits of integrating SAS model governance into an analytical ecosystem that includes a diverse set of open-source tools and many different scripting languages next to SAS analytics. Specifically, the customer asked us to describe a continuous integration process for operationalizing analytical models in this environment. In this series of posts, we want to share some of our experiences with you.

Before we dive into the details, let me clarify that the term “model validation” is used in a rather narrow sense here to describe an approving process for analytical models before they are accepted for deployment to a production environment. In this case, since the models will be implemented as part of a CI process, it's essential that this “pipeline” has to be fully automated.

In total, there are four parts to this blog post series that describe this modeling scenario in detail:

1. This post will describe some basic principles of the DevOps (or ModelOps) approach.
2. [The second post](#) discusses the “test pyramid,” which originally is an industry-standard for test design in software engineering. However, we think it's valuable in the area of analytical test design as well.
3. The [third post](#) is more “hands-on” in nature. It describes how a model validation pipeline can be implemented in real life – using tools like Git, Jenkins – and SAS Model Manager of course.
4. Finally, the [fourth post](#) addresses a purely analytical topic. While the first three posts are written with a DevOps engineering audience in mind, the fourth post is aimed more at data scientists. It contains a detailed explanation of an algorithm called the feature contribution index (FCI), how it works and how it can be used with SAS. This algorithm has been essential in the project and we hope you will find that the FCI could be a valuable tool in your analytical toolbox too – whether your aim is to set up a “model validation pipeline” or something completely different.

So, let's get started. Have fun!

Back to the basics: some DevOps principles

Continuous automation is a popular DevOps method for automatically building, testing and validating code every time a change is made from anyone on a development team. This makes model test automation a typical natural integration use case. While continuous integration is often discussed in the context of DevOps, we're applying the method in ModelOps. Of course, the idea of ModelOps did not appear out of thin air. In fact, its core principles go back to the base characteristics of the DevOps approach. The most important of them being automation, test first and API first.

Automation

By automation we're referring to both deployment automation and test automation. While the latter is in the focus of this blog, deployment automation is a crucial aspect of CI as well. The latest release of [SAS Viya](#) brings a strong focus on container technology, which is typically a good answer to the automation challenge, as [containers](#) provide a stable runtime environment for analytical models.

The container becomes the deployable unit that is used to safely ship a model through the environments that are commonly seen as enterprise deployment architectures (development, testing, production, etc.). The current version of [SAS Model Manager](#) is a prime example of this principle as it leverages container technology for analytical models written in an open source language like Python and R.

Essentially, Model Manager packages together all the needed ingredients before pushing the model container to a Kubernetes cluster: the Python or R kernel, all client required packages, and finally the analytical model as well. (And by the way: you are on the right track if this process reminds you of the source-to-image (s2i) build mechanism in Kubernetes orchestration platforms such as Red Hat OpenShift.)

In this blog post, we're less concerned about the particular deployment technology in use, but clearly containers fit very well into the concept of automated validation. Using container technology, we can make sure that we are validating the exact artifact that will then eventually be pushed to a production environment. The benefits include no version glitches, no side effects and easy to keep under version control as a whole for auditing purposes.

Test first

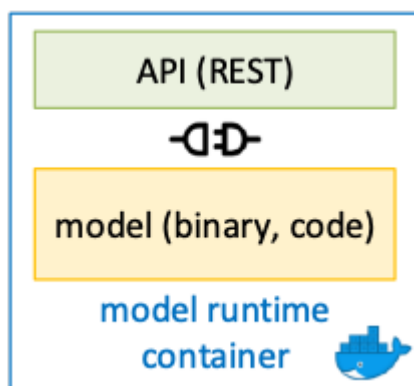
The test-first paradigm tries to pinpoint the best spot in the lifecycle of software development (or model development in this case) when testing should occur. Test-driven development is a standard today in the field of software engineering. Developers are asked to first write their tests (often known as unit tests) before they start writing the actual production code. The idea here is that you have your unit tests as a safety net in your back from the very beginning on.

Testing becomes a background activity that is continuously executed whenever code changes are detected and, as such, it needs to be as non-obstructive as possible: as long as tests do not fail, developers should not even take notice of them at all. It is easy to see how this requirement relates to the previous point we were talking about: test automation.

So how would this test-first concept translate to the analytical space? Quite easily. In fact, but with minor modifications, since model validation adds a quality perspective to testing. Model testing not only covers simple "pass/no pass" checks but also keeps an eye on, "how well did this model pass?" and "how does this model compare to others?" In other words: the prediction quality of the model is probably the most important validation step we should focus on.

API first

Probably being less popular than the previous one, the "API first" pattern rapidly gains in importance for any non-trivial software and especially proves its worth over time – when changes creep in and software needs to adapt to new requirements. "API first" fosters a modular approach to architecture as it asks developers to first design the public interface of a component before moving on to the implementation. This public interface now becomes a stable "contract" on which other components can rely, unaffected of any changes going on the implementation level over time.



As before, the “API first” paradigm can be found in the area of analytical work as well as the area of DevOps. To give an example: imagine a real-time scoring model that exposes its functionality via a REST-based interface. A web shop system is relying on this service during the checkout stage to make a choice on the payment methods it will allow.

As we know, the prediction quality of analytical models usually degrades over time, so at some point, the scoring model might need to be retrained or even replaced by a different model implementation. In this situation, two requirements become essential: first, the client of the scoring service should not be impacted by the necessary back end changes at the model implementation level (what happens in Vegas, stays in Vegas). Second, the update better not cause any downtime.

Most container orchestration platforms offer a rolling update deployment strategy which will take the latter requirement. When taken seriously, “API First” should ensure that the public interface of the retrained or rewritten model has not changed (so clients can still use it). But there’s more to it when looking “into” the runtime container. In most cases, the API is actually an “API server” – an active layer in the container acting as the exposed communications endpoint – and it is equally important to check that the updated model still successfully communicates with that API server.

What’s next?

In this blog post, we described how the base characteristics of the DevOps approach – automation, test first and API first – can be adapted to the area of analytical work. In the [next post](#) in this blog series, we will take a closer look at the test design topic. Is there a neat way of ordering and separating different categories of tests and how would that look if applied to analytics? Check it out and thanks for reading!

Model validation testing in the age of DevOps

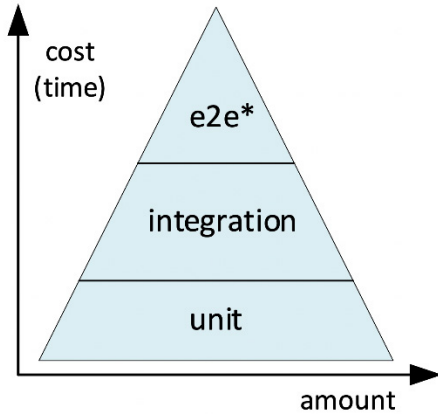
By Hans-Joachim Edert

Published on The SAS Data Science Blog, February 19, 2020

This blog post is part two of a [series on model validation](#). The series is co-authored with my colleague [Tamara Fischer](#).

After revisiting some of the key principles of DevOps and discussing how to map them to the area of analytical work in the [first post](#) of this series, let us now take a look at a well-known metaphor for test case development in the software industry. We are referring to the idea of the “test pyramid” ([see here for a thorough almost canonical explanation by Martin Fowler](#)).

There are many variations of the test pyramid as it is not a strict methodology. It's simply a graphical representation that helps to get a clear view on the constraints to be taken into account when designing test cases: how they can be grouped (low-level to high-level), how many of them (in relation to the total) are to be expected, and the costs associated to them. Our version of the test pyramid looks like this:



Our test pyramid depicts our vision for analytical model testing (*e2e = “end-to-end”)

Let’s quickly walk through the layers to see how these sections of the pyramid can be applied to the validation of analytical models.

Unit tests

The bottom layer of our test pyramid is composed of unit tests. Unit tests are small, low-level checks aiming at validating infrastructure and coding policies. Typical unit test cases would be checking if the developer has committed all required files (model binary package, training data, metadata descriptors etc.) or checking if the code adheres to policies and conventions (has a header, has comments, etc.). Unit tests typically are quickly executed and can easily be reused.

For example, the following SAS code checks if the developer has included the model to the project before it was committed. It is assumed that the model function is stored in a single file, usually this is a file in the ASTORE format (analytical store). The name of the model file (passed to this test in the SAS macro variable MODELFILE) is specific to the project, so it is not part of the test but the test, as such, can be easily reused between any project.

```

/* ***** */
/* Unit Test: Test if the requested model file is available */
/* ***** */
%include "/tmp/driver.sas";

options nomprint nosource;
%MACRO CHECK_MODEL;
  %put Checking for model file &MODELFILE. ;
  %if %sysfunc(fileexist(/tmp/&MODELFILE.)) %then %do;
    %put SUCCESS: The file &MODELFILE. was found. ;
  %end; %else %do;
    %put ERROR: The file &MODELFILE. does not exist. ;
  %end;
%mend;
%CHECK_MODEL;
options mprint source;

```

You might wonder about the return codes. Shouldn’t the test cancel or abort the SAS session if it fails? Is it sufficient to print out a simple “ERROR” statement to the log? In fact, it is – at least in this case. We’ll get back to this point at the end of this post.

A final remark at this point: if you're searching for more sophisticated approaches for defining unit tests for SAS (our example given above is admittedly super simple), make sure to read recent SAS Global Forum papers like [this one](#) or take a look at this [unit test framework hosted at SourceForge](#).

Integration tests

This group of tests checks how well the analytical model fulfills its purpose outside the training sandbox, where it will deal with data of a different quality (missing attributes, corrupt records) and frequency (like streaming data). Integration tests aim higher when compared to unit tests, because they try to measure model quality and model performance before the model is actually deployed to production.

Integration tests are also quite sophisticated in nature, as they use analytical algorithms to measure other analytical algorithms. One clever test falling into this category is the [Feature Contribution Index](#), which analyzes the covariance structure of the predictors only. The advantage of this test is that a target variable is not needed to evaluate whether a model can be applied to new data.

If you're interested in a detailed description of how the analytics behind this test works, take a look at this [blog post](#).

End-to-end tests

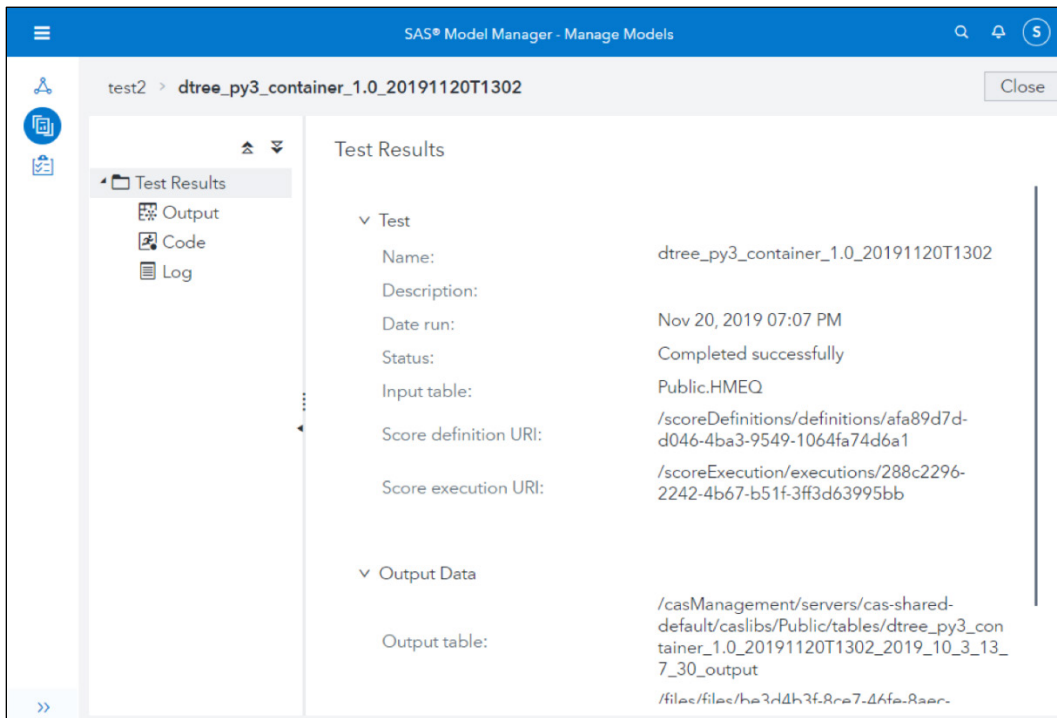
The final group of tests could also be renamed to API tests. These tests try to uncover defects that only can be seen in the "end-to-end" scenarios that include the full loop from the client making the request to the service returning the response. One potential consequence is that true e2e tests might not be suitable for full automation. Testing the API, however, can be fully automated.

What's the purpose of testing an API? Again, it is primarily a counter-measure to shield against changes creeping in over time.

To illustrate one potential scenario for model validation, take a look at the API defined by the model container images we're using at SAS (here's the [GitHub project](#) with the source codes). These containers, meant for executing analytical models written in Python or R, support a method call to trigger a scoring operation (taking the data to be scored as input). SAS Model Manager uses this API for interactively testing / validating the model using the graphical user interface.

The screenshot shows the SAS Model Manager interface for a model named 'test2'. The 'Tests' tab is active, displaying a table of test results. The table has columns for Name, Results, Status, Date Last Run, Model Name, Project Version, and Target Destination. The tests are for different versions of the 'dtree_py3_container' model, with versions 1.0, 2.0, 3.0, and 5.0 showing successful results (green checkmarks), while versions 4.0 and 6.0 show failed results (grey circles).

Name	Results	Status	Date Last Run	Model Name	Project Version	Target Destination
<input type="checkbox"/> dtree_py3_container 1.0 2...		✓	Nov 20, 2019 07:07 PM	dtree-py3-container (1.0)	Version 1 (1.0)	PrivateDockerDemo
<input type="checkbox"/> dtree_py3_container 2.0 2...		✓	Nov 21, 2019 09:46 PM	dtree-py3-container (2.0)	Version 1 (1.0)	PrivateDockerDemo
<input type="checkbox"/> dtree_py3_container 3.0 2...		✓	Nov 29, 2019 04:01 PM	dtree-py3-container (3.0)	Version 1 (1.0)	PrivateDockerDemo
<input type="checkbox"/> dtree_py3_container 4.0 2...		●		dtree-py3-container (4.0)	Version 1 (1.0)	PrivateDockerDemo
<input type="checkbox"/> dtree_py3_container 5.0 2...		✓	Nov 29, 2019 04:05 PM	dtree-py3-container (5.0)	Version 1 (1.0)	PrivateDockerDemo
<input type="checkbox"/> dtree_py3_container 6.0 2...		●		dtree-py3-container (6.0)	Version 1 (1.0)	PrivateDockerDemo



However, it's easy to see that the same method call can be called in a batch pipeline as well.

Working on a chain gang

Coming back to a point we discussed earlier: what's the appropriate response if one or more tests fail? As you've noticed in the simple example given above, we decided to not immediately cancel all processing. Instead we're simply printing out the "test failed" notification to the SAS log. Why is that?

It's important to understand that each test is only one out of potentially many being executed by an automated pipeline outside the immediate reach of the developer, which means it is crucial to aggregate the test results and send them back to the developer.

In our case (as you'll see in the [3rd part](#) of this blog), we decided that we want to separate these steps in the automation pipeline: first we run all tests, push all generated output (logs, reports) back to a Git system and *then* evaluate the test results (stopping the pipeline at this point if we detect that a test had failed before). We believe that this is a more efficient way of sending feedback to the developer instead of sending separate notifications about each test (especially if there is more than one test failing).

We're using Jenkins as the test automation system, and a pipeline written in Jenkins (the so-called "Jenkinsfile") is basically a JSON document containing one or more (Linux) shell scripts sequentially chained together in stages. The following snippet should give you an idea of how we organized the validation pipeline. It shows the three most important stages:

1. Sequentially run all tests. Each test is kept in a separate SAS file following a naming convention (test*.sas).
2. Commit the output of all tests back to the original Git project.
3. Sequentially evaluate the test logs (test*.log). Cancel processing if one of the tests has returned an error.

```

1. stage('Run unit tests and quality checks') {
2.   steps {
3.     sh '''
4.       for f in test*.sas
5.       do
6.         docker exec $myCnt \
7.           su -c "/opt/sas/spre/home/bin/sas -SYSIN /tmp/$f \
8.             -CONFIG /opt/sas/spre/home/SASFoundation/sasv9.cfg \

```

```

9.         -CONFIG /opt/sas/spre/home/SASFoundation/nls/u8/sasv9.cfg \
10.        -PRINT /tmp/results.lst \
11.        -LOG /tmp/results.log" testrunner
12.
13.        (docker exec $myCnt cat /tmp/results.log) &gt; $f.log
14.        (docker exec $myCnt cat /tmp/results.lst) &gt; $f.lst
15.    done
16.    '''
17. }
18. }
19. stage('Commit test results back to git project') {
20.     steps {
21.         ... code omitted ...
22.     }
23. }
24. stage('Evaluate test results') {
25.     steps {
26.         sh '''
27.             for f in test*.log
28.             do
29.                 # grep for "ERROR", return false if found
30.                 cat $f | if [ $(grep -c "ERROR:") == 0 ]; then exit 0; else exit 1; fi
31.             done
32.         '''
33.     }
34. }

```

What's next?

After reviewing some of the basic DevOps principles in the [first post](#) of this series, we introduced the concept of the test pyramid in this installment, which helps us to organize the test cases we want to run against the analytical models. The [next part](#) will discuss the overall system architecture and will also share more details on the infrastructure (Jenkins, Docker etc.) that we used. Stay tuned!

Creating a model validation pipeline

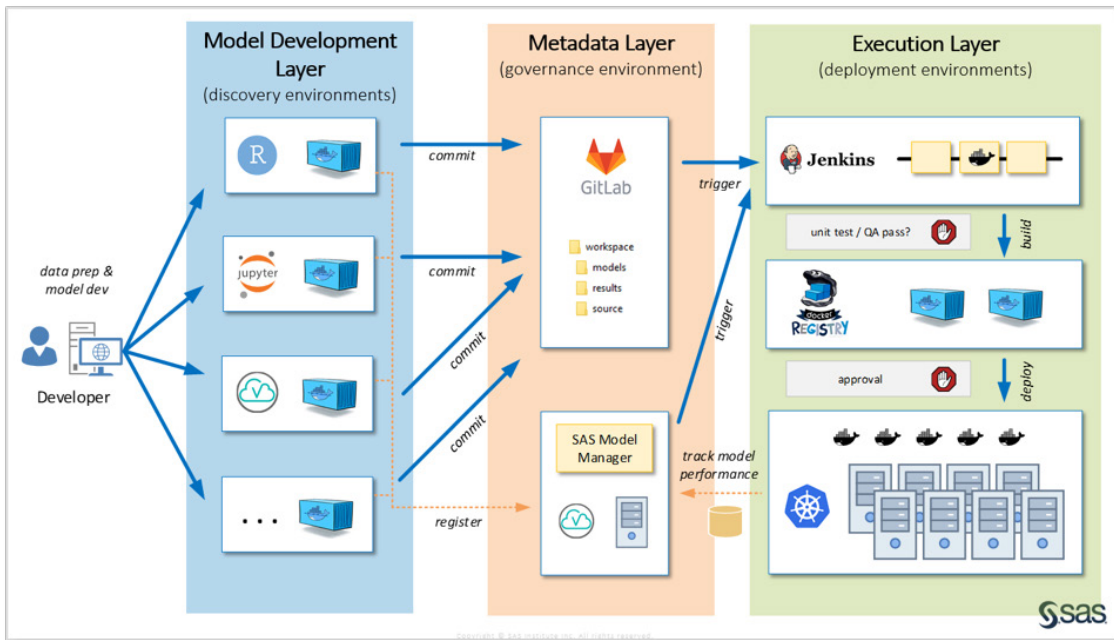
By Hans-Joachim Edert

Published on The SAS Data Science Blog, February 19, 2020

This blog post is part three of a [series on model validation](#). The series is co-authored with my colleague [Tamara Fischer](#).

So far, we've been discussing [how the base characteristics of the DevOps approach can be applied to the area of analytical work](#), and we talked about [different test categories and how to chain them together](#). Let's now move from theory to process. Using an analytical platform we recently set up for a SAS customer as our example, this post will describe a fully automated validation pipeline for analytical models.

To get started, here’s a simplified overview of the main components of this platform:



Let's briefly discuss the different components.

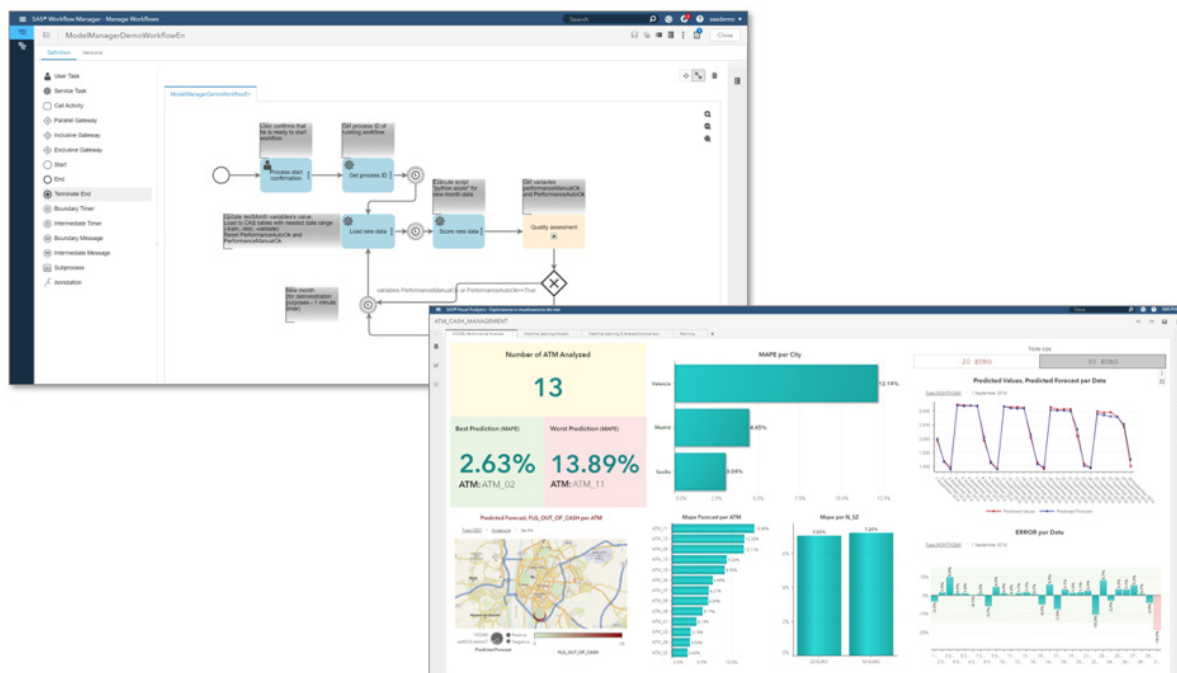
Model development (blue)

Data scientists working in this environment decide which modeling tools they want to use, such as R, Python or SAS. Container technology is used to provide individual work spaces for them. [SAS Studio](#), Jupyter Notebook and R Studio are the web-based code editors used by the data scientists. Note that this approach usually requires an “umbrella workbench” application, which is the primary interaction endpoint for users and provides ways to handle the container life cycle. In other words, the work bench offers a GUI with an easy way for users to launch and stop their workspaces, etc.). These workbenches could be developed in-house, but there are also [commercial products](#) available which provide a seamless integration with SAS container technology.

Metadata (orange)

As containers are ephemeral in nature, all project artifacts, such as program files, documentation or even small amounts of training data, are kept in a Git source code management system. However, in addition to that, analytical models (e.g., pkl files, SAS ASTORE files) are registered in [SAS Model Manager](#), which is used as a centralized model repository.

Unlike Git, SAS Model Manager is a domain specific tool which was specifically designed to manage analytical models. It provides additional functionality on top of just storing the file artifacts, such as workflow management, model performance monitoring and reporting (shown in the screenshots below).

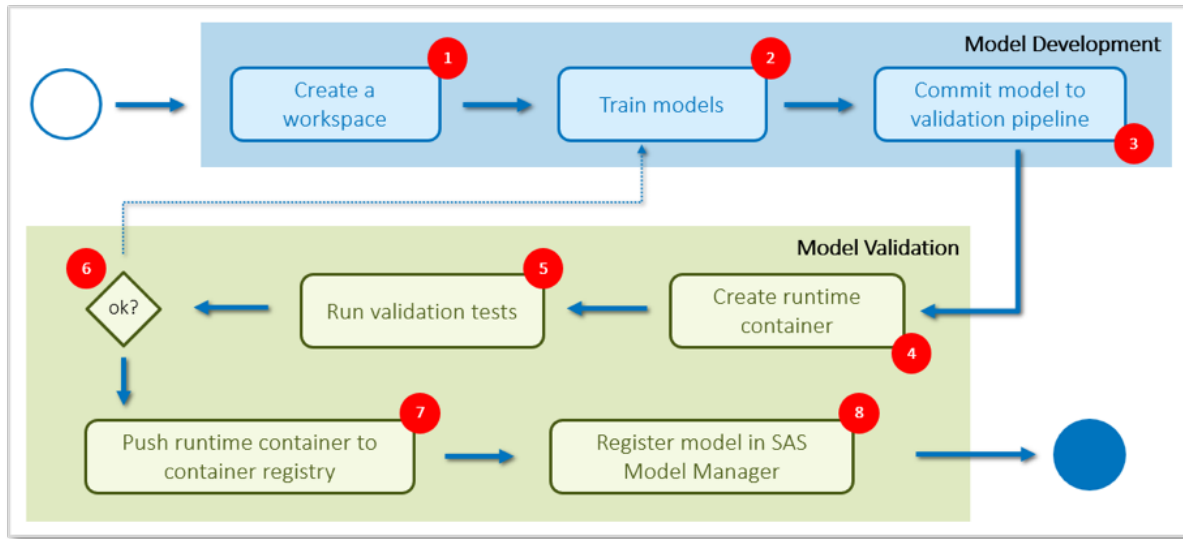


In the real-world example we're using for this blog, SAS Model Manager is an endpoint as the pipeline feeds the validated models into it. However, when taking the wider perspective of looking at the full life cycle of analytical models, SAS Model Manager is also a starting point. From within SAS Model Manager, you can publish a model to its production environment (be it a database, a Hadoop system, a web service or a container orchestration platform), and it can also be used to collect feedback about your model's performance – allowing you to decide whether it's necessary to retrain (or replace) the model.

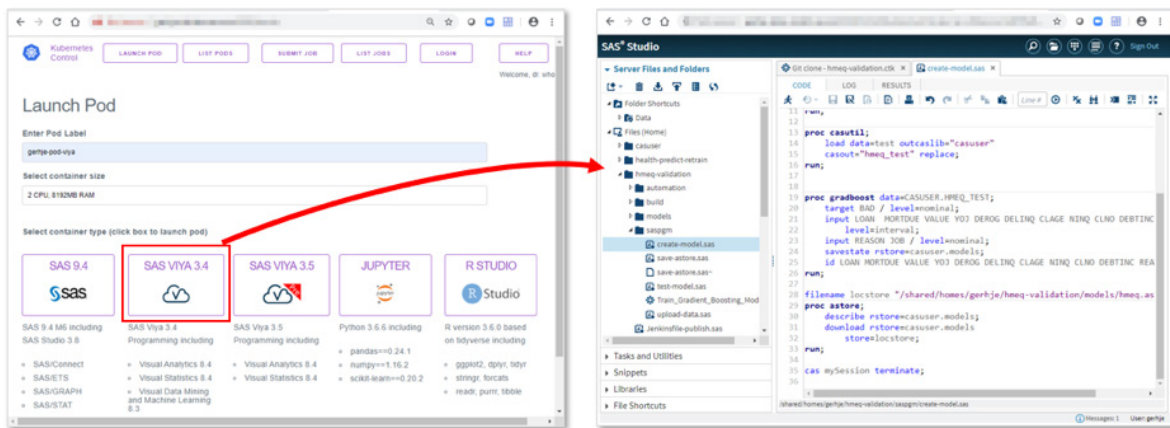
Execution (green)

The execution layer consists of a Kubernetes cluster providing an environment for model runtime containers, as well as for the interactive individual workspace containers. Next to Kubernetes, a Jenkins CI server is configured to run multiple pipelines. These pipelines can be triggered interactively (by pushing a button in SAS Model Manager) or by changes (commits) to the Git projects. Validation pipelines will launch model containers on the Kubernetes cluster and run preconfigured test cases on these models.

These components are connected to each other to create an automated validation pipeline for analytics models. The basic idea is shown in the following workflow diagram:



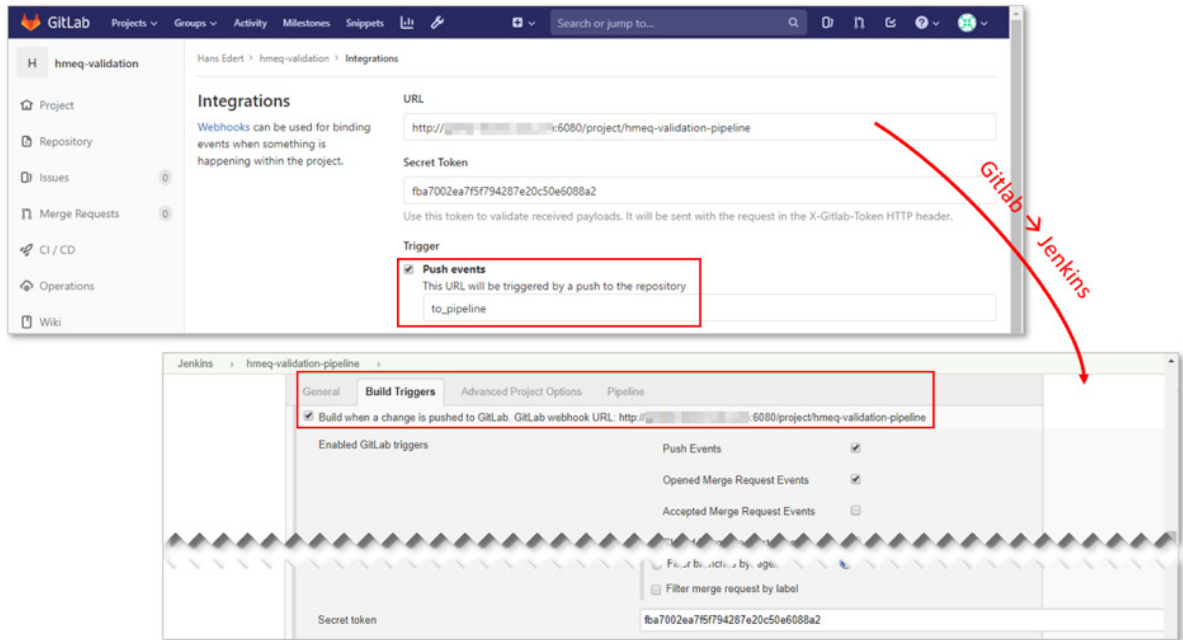
(1) Data scientists begin by requesting the individual workspace they need. For example, they could launch a [SAS Viya](#) development environment:



(2) While working on their tasks, the data scientists interact with a Git SCM system, which is the central source repository for all project files (for example, source codes, training data or documentation). A milestone is reached once the user has completed his work up to a point where he or she thinks that the analytical model is ready to be committed to the validation pipeline (3).

At this point an automated process kicks in. The pipeline acts like a quality gateway to make sure that the outcome of the developer’s work is eligible to be registered to the model repository (in this case SAS Model Manager). From there, it can then be published to a production environment. To make sure that the model meets all the requirements, the pipeline runs a collection of unit, integration and API tests against it. Usually these tests are written in the same language that was used for model development (refer to the [previous blog post](#) for a simple example of a unit test written in the SAS language).

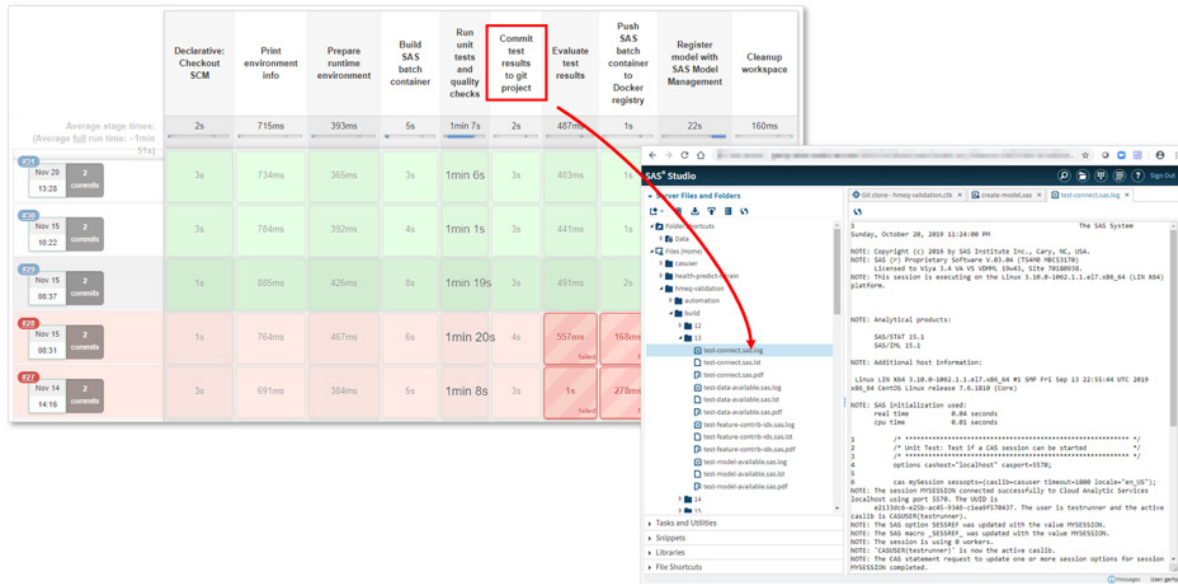
As we mentioned in post two, ModelOps testing should be automated and non-obstructive. From a technical perspective this is achieved by integrating an automation server with the SCM system. In our case, we're using Jenkins and Git, but there are certainly alternatives to both tools. Jenkins listens to changes in the Git projects and will trigger the pipeline execution once a user checks in his work to a certain Git branch of his or her project – speaking technically: We've defined a webhook:



In the previous post we've also mentioned end-to-end or API tests. What makes these tests valuable is that they usually run in environments which are close to production. And again, container technology provides a big advantage at this point; the model runtime container image is our deployable unit which can be tested in semi-production and pushed to a production environment without being re-built or re-deployed. So, at the beginning of the validation pipeline, a runtime container image will be set up (4). It consists of the suitable language kernel (e.g. SAS, R, Python – check [here](#) for more details) and the specific model which has been committed to the pipeline. All checks will be run against this container image (5) and if successful (6), this image will be pushed to a container registry (7). Refer to our [previous blog post](#) for a more detailed discussion on the sequential steps of the Jenkins pipeline.

12 SAS and Open-Source Model Management

As a final step, the model will also be registered in SAS Model Manager (8) using the collection of [Model Management macros](#) which are supplied out-of-the-box with SAS Model Manager. Here's a screenshot showing the Jenkins CI server executing a validation pipeline after the user has submitted the analytical model to Git:



This picture shows successful and failed pipeline executions. The pipelines highlighted in red did fail for at least one test, and as a consequence neither the container image was pushed to the registry nor was the model registered in SAS Model Manager. Follow the column headings to get an idea of the various stages of the validation pipeline. It's worth mentioning that the pipeline is set up so that developers receive timely feedback when commits are being rejected – which is another core requirement for CI workflows. For example, error logs, listings or even PDF documents generated by the unit tests are pushed to the model's Git project by the automation server, so they're available for developers in their working environment.

Conclusion

We've tried to cover a lot in this blog series, but there is still so much more we could say. For example, you might wonder: What happens to the analytical models that passed the acid test of the model validation pipeline and made it into the image repository? After all, they're just at the beginning of their life cycle at this point! We do have good answers to these questions (hint: SAS Model Manager is a good start), but this is a story for another blog post.

In this and the previous two blog posts, we've described a way to validate analytical models adhering to continuous integration best practices up to the point where they become ready for the "go-live". Automation (avoiding human intervention), test first and API first have been the guidelines for this process. And while we all probably do agree on the importance of the CI approach as such, it's also clear that every implementation will be different. So hopefully these blogs have given you some inspiration, and we'd be happy to learn about your approach to operationalizing analytics, a.k.a. completing "the last mile."

Introducing the feature contribution index for model assessment

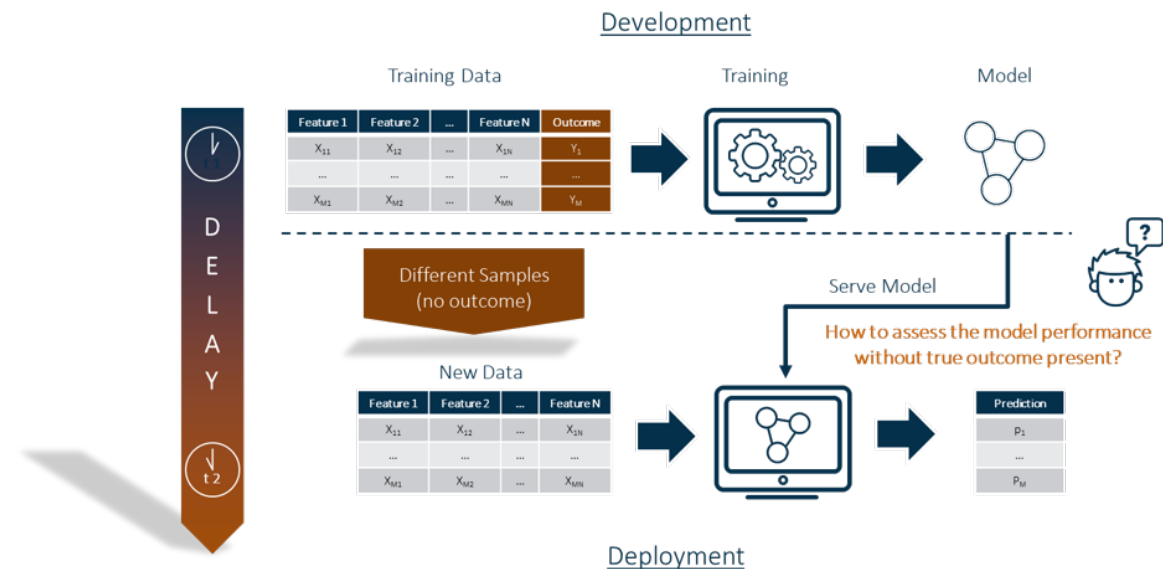
By Tamara Fischer

Published on The SAS Data Science Blog, February 19, 2020

This blog post is part four of a [series on model validation](#). The series is co-authored with my colleague [Hans-Joachim Edert](#).

Most model assessment metrics, such as lift, area under the curve, Kolmogorov-Smirnov statistic or average square error, require the presence of the target/label to be in the data. This is always the case at the time of model training. But how can I ensure that the developed model can be applied to new data for prediction? There may be weeks or even months between model development and model deployment. This means that the distribution of the predictors/features may have changed. Even if model development and model deployment happen quickly, the training data may differ from the new data by sampling (see Figure 1).

Figure 1: Different samples of training and new data due to delay in time and absence of the true outcome

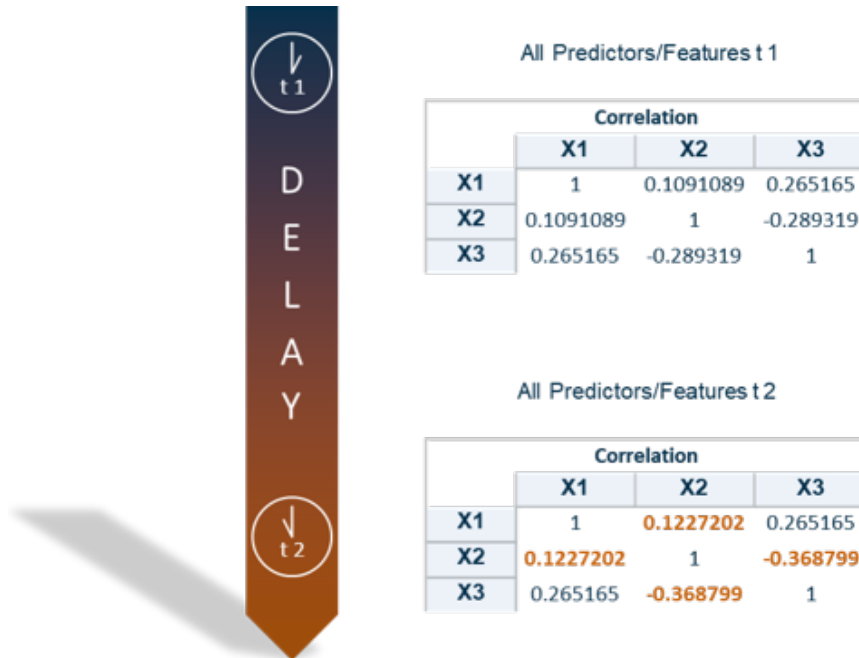


The true outcome, or target/label, is usually not available in the data at the time of model deployment. This means that the usual model assessment metrics cannot be used for model assessment.

The calculation of a feature contribution index allows you to evaluate a model without the presence of a target variable. That makes it suitable for use as an analytical test in a ModelOps scenario, which can be automated. If you're interested in learning more, check out the other posts in our [blog series on model validation](#).

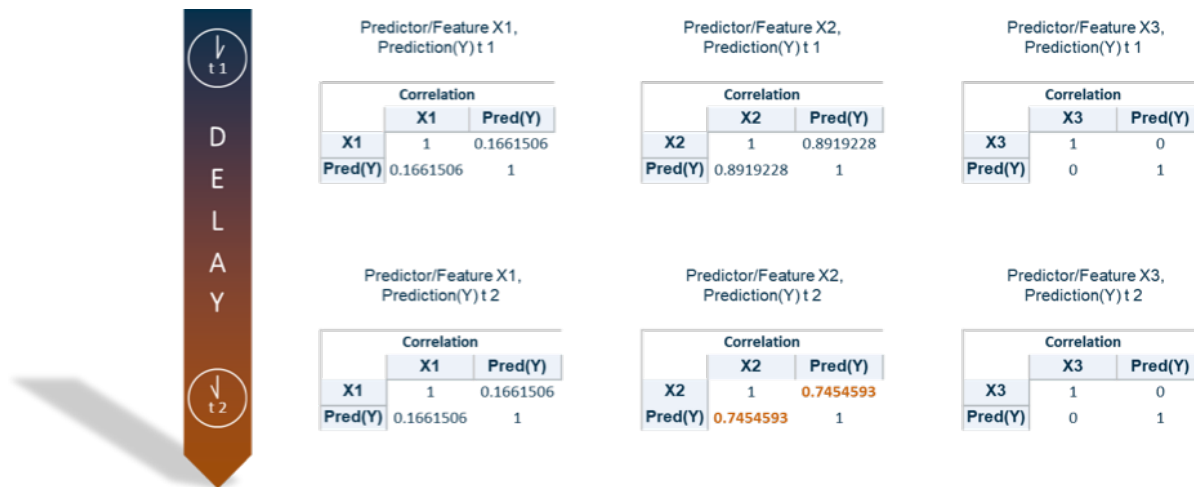
Let's dig deeper into the idea of the feature contribution index. It's based on the idea that the application of a model to new data is permissible if the associations (both strengths and directions) among the predictors/features of training and new data are similar. We used the correlations to measure the associations. This has the advantage that the values are all between -1 and 1 (see Figure 2).

Figure 2: Correlation matrix of all predictors



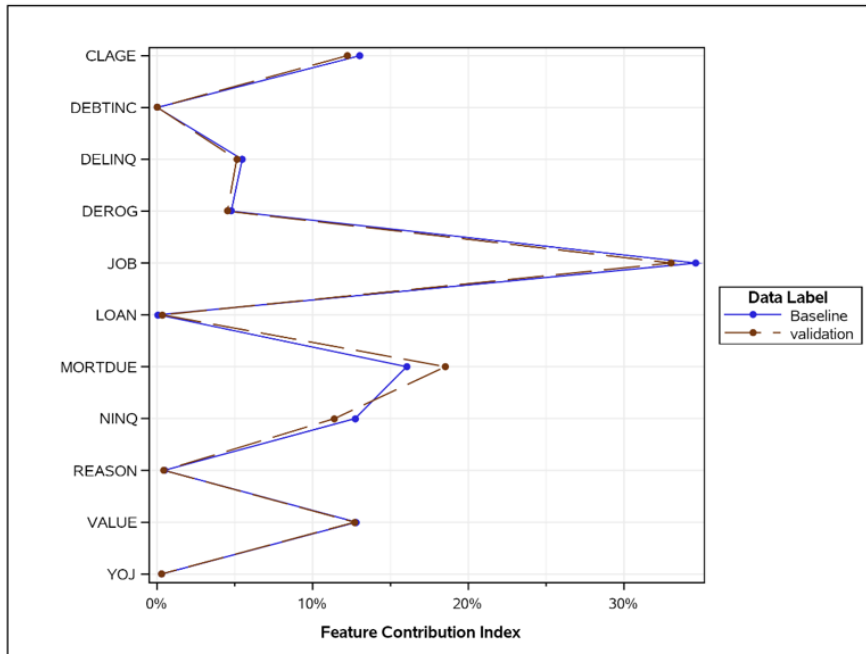
In order to make a statement about deviations for each predictor/feature, the correlation between each predictor/feature and the prediction can alternatively be calculated (see Figure 3).

Figure 3: Correlation matrix between the prediction and each predictor



SAS® Model Manager includes SAS macros to calculate the contribution index for each feature. Below you can see an example where the feature contribution index of some predictors/features are plotted for two points in time. You can see that the deviation of the predictor “MORTDUE” is the largest here (see Figure 4). But is it **too** large?

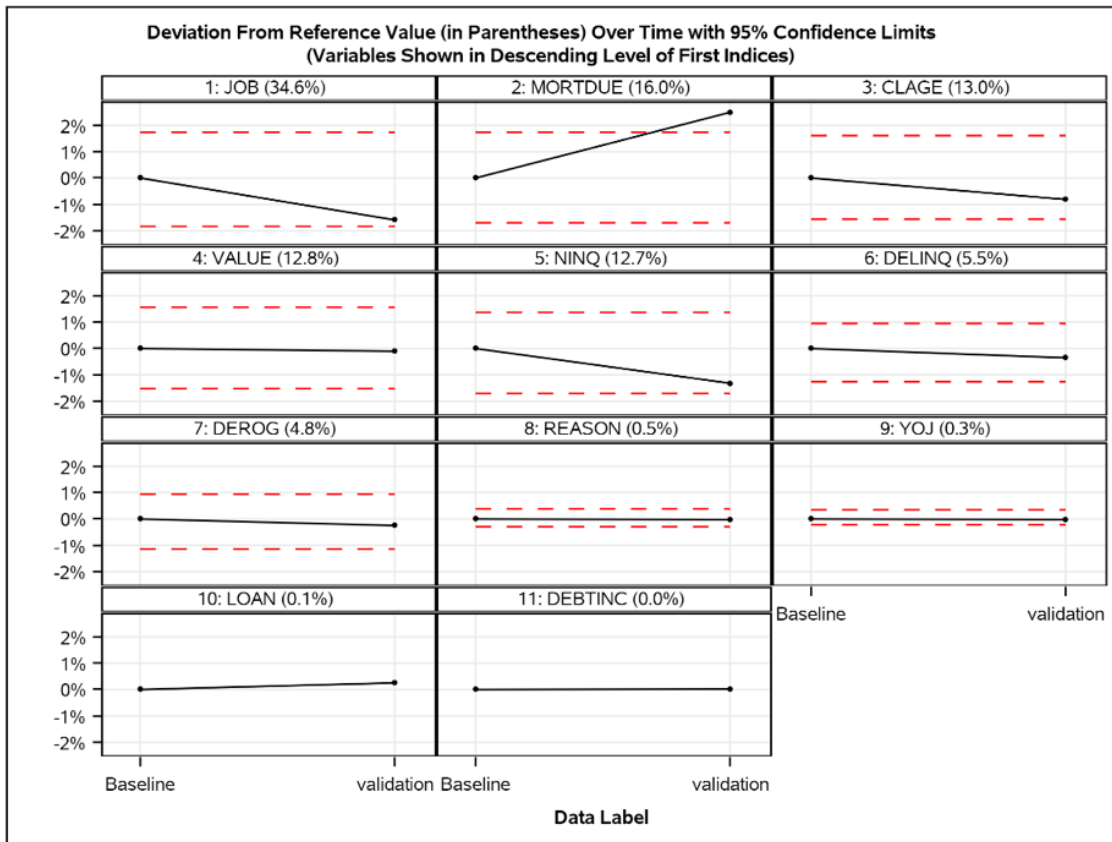
Figure 4: Feature contribution index for two points in time: development (baseline) and deployment time (validation)



So, now we have to find a way to define the “similar”? How large may the deviation be? Because some random elements are always present in the data and should be allowed. This can be achieved by calculating confidence bands whose limits should not be exceeded, using the Baseline values as the references. The details on how the confidence bands are calculated are beyond the scope of this blog, but can be found in this SAS Global Forum Paper [“Monitoring the Relevance of Predictors for a Model Over Time,”](#) authored by Ming-Long Lam, Ph.D., who works in R&D at SAS.

In Figure 5 below, only the predictor “MORTDUE” exceeds the confidence limit. All other predictors are within the confidence limits.

Figure 5: Deviation confidence bands for each variable with calculated feature contribution index for two points in time: development (baseline) and deployment time (validation)



With SAS Model Manager 15.3 on Viya, the Feature Contribution Index is available with each model monitoring report request. If you’d like more information about SAS Model Manager, visit our [Help Center](#).

APPENDIX

For further reading, we suggest the following resources.

Whitepapers, e-books and SGF papers (downloadable)

Ball, T. and Hughes, C. (2020). [Open Source Python & R Lang on our SAS® Shared Grid](#)

Mohan, P. (2020). [Practical Geospatial Analysis of Open and Public-Use Data](#)

SAS e-book. [Getting started with ModelOps](#)

SAS e-book. [Out in the open with analytics](#)

SAS e-book. [SAS in the Open Ecosystem – How a unifying platform can bring together diverse data and analytics to drive measurable value for your organization](#)

TDWI Pulse Report (2020). [Using a Hybrid Open Source and Commercial Analytics Ecosystem](#)

Thangamuthu, K. (2019). [Sparking Your Data Innovation: SAS® Integration with Apache Spark](#)

Toporowski, A. (2020). [Python and the SAS® Quality Knowledge Base for Better Data Quality and Entity Resolution](#)

On-demand webinars and video contents

Burgess, D. and Long, J., [Automating Model Operations End to End](#)

Furbee, J. [Using SAS APIs](#)

Hart, K., Jensen, B.K., and Walker, M. [Make Your Models Move \[ut. Deploy your analytical models whenever and wherever needed](#)

Long, J. and Ching, Y.J., [Easily deploy and manage your models \(R, Python or SAS\)](#)

Malley, P., and Mohammad, M., [ModelOps: Operationalising Analytics](#)

Mendes, N. and Long, J., [Identifying Model Drift Before It Is Too Late](#)

SAS Demo – [SAS Open Model Manager. First step deploying open source models](#)

SAS Users – [Open-Source Model Management with SAS® Model Manager](#)

Blogs and articles

Edert, H-J., [Creating a model validation pipeline](#)

Newell, J., [Driving faster value from analytics – how to deploy models and decisions quickly](#)

Ochiai-Brown, J., [The last mile – getting analytics into operations](#)

Vandenbergh, F., [How to deploy your models with SAS Model Manager to Hadoop](#)

Ready to take your SAS[®] and JMP[®] skills up a notch?



Be among the first to know about new books,
special events, and exclusive discounts.

support.sas.com/newbooks

Share your expertise. Write a book with SAS.

support.sas.com/publish

 sas.com/books
for additional books and resources.


THE POWER TO KNOW.®

