Paper SAS1492-2017

An Overview of SAS® Visual Data Mining and Machine Learning on SAS® Viya

Jonathan Wexler, Susan Haller, and Radhikha Myneni, SAS Institute Inc., Cary, NC

ABSTRACT

Machine learning is in high demand. Whether you are a citizen data scientist who wants to work interactively or you are a hands-on data scientist who wants to code, you have access to the latest analytic techniques with SAS® Visual Data Mining and Machine Learning on SAS® Viya. This offering surfaces in-memory machine-learning techniques such as gradient boosting, factorization machines, neural networks, and much more through its interactive visual interface, SAS® Studio tasks, procedures, and a Python client. Learn about this multi-faceted new product and see it in action.

INTRODUCTION

Solving modern business problems often requires analytics that encompass multiple algorithmic disciplines, data that is both structured and unstructured, multiple programming languages, and – most importantly – collaboration within and across teams of varying skill sets. Addressing and solving business problems should not be constrained by technology. Technology enables analysts to solve problems from multiple angles. Likewise, computing power is cheap. Problems that were once deemed unsolvable using neural networks can now be run in mere seconds.

This paper shows you how to solve business problems, quickly and collaboratively, using SAS Visual Data Mining and Machine Learning on SAS Viya. This new offering enables you to interactively explore your data to uncover 'signal' in your data. Next you can programmatically analyze your data using a rich set of SAS procedures covering Statistics, Machine Learning, and Text Mining. You can add new input features using in-memory SAS DATA step. Utilize new tasks in SAS Studio on the SAS Viya platform to automatically generate the SAS code. If you prefer to write Python, access SAS Viya methods with the Python API. No matter the interface or language, SAS Viya enables you to start your analysis and continue forward without any roadblocks.

In this paper, you will learn how to access these methods through a case study.

SAS VISUAL DATA MINING AND MACHINE LEARNING ON SAS VIYA

SAS Viya is the foundation upon which the analytical toolset in this paper is installed. The components are modular by design. At its core, SAS Viya is built upon a common analytic framework, using 'actions'. These actions are atomic analytic activities, such as selecting variables, building models, generating results, and outputting score code. As shown in Figure 1, these actions can be accessed via SAS procedures, SAS applications, RESTful services, Java, Lua, and Python.

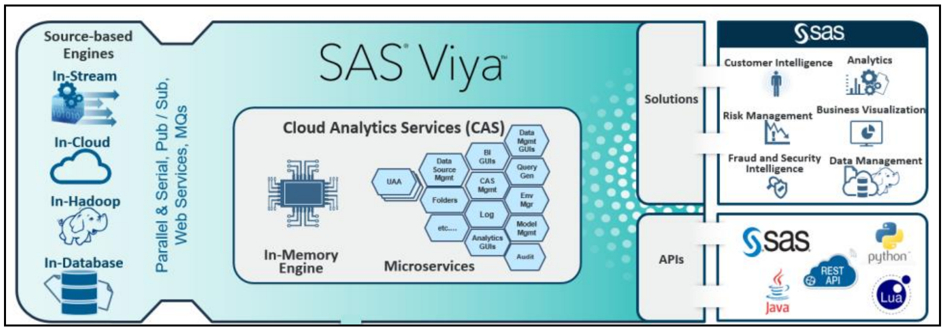


Figure 1. SAS Viya Ecosystem Is Open and Modular

SUPPORTED SAS VIYA ALGORITHMS

From a data mining and machine learning perspective, SAS Visual Data Mining and Machine Learning on SAS Viya enables end-to-end analytics - data wrangling, model building, and model assessment.

As shown in Table 1, the following methods are available to users:

Data Wrangling	**Modeling**
Binning	Logistic Regression
Cardinality	Linear Regression
Imputation	Generalized Linear Models
Transformations	Nonlinear Regression
Transpose	Ordinary Least Squares Regression
SQL	Partial Least Squares Regression
Sampling	Quantile Regression
Variable Selection	Decision Trees
Principal Components Analysis (PCA)	Forest
K-Means Clustering	Gradient Boosting
Moving Window PCA	Neural Network
Robust PCA	Support Vector Machines
	Factorization Machines
	Network / Community Detection
	Text Mining
	Support Vector Data Description

Table 1. Analytic Methods Available in SAS Visual Data Mining and Machine Learning on SAS Viya

You will experience increased productivity when using the aforementioned methods. All of these methods run in-memory, and take advantage of the parallel processing ability of your underlying infrastructure. The more nodes you have; the higher degree of parallelism you will experience when running. Once data is loaded to memory up-front, you can run sequential procedures against the same table in memory, eliminating the need to drop the data to disk after each run. You can continue your analysis using the same data in-memory. If the memory of your problem requires more memory than is available, the processing will continue over to disk.

There were numerous analytic innovations that we introduced with SAS Viya. At the head of the class is hyperparameter autotuning (Koch, Wujek, Golovidov, and Gardner 2017). When data scientists tune models, they train the models to determine the best model parameters to relate the input to a target. When they tune a model, they determine the architecture or best algorithmic hyperparameters that maximize predictability on an independent data set. Autotuning eliminates the need for random grid search or in a SAS user's case, running repetitive procedure calls with different properties. As shown in Figure 2, Autotuning uses a local search optimization methodology to intelligently search the hyperparameter space for the best combination of values that addresses the model objective – that is, misclassification, Lift, KS, and so on. Autotuning is available for Decision Trees, Neural Networks, Support Vector Machines, Forests, Gradient Boosting, and Factorization Machines.

Also new in SAS Viya are enhanced feature engineering techniques like Robust PCA (RPCA), Moving Window PCA, and the capability to detect outliers using Support Vector Data Description (SVDD). Robust PCA decomposes an input matrix into low-rank and sparse matrices. The low-rank matrix is more stable as the distortions in the data are moved into the sparse matrix, hence the term robust. Moving Window PCA captures the changes in principal components over time using sliding windows and you can choose RPCA to be performed in each window. SVDD is a machine learning technique where the model builds a minimum radius sphere around the training data and scores new observations by comparing the observation's distance from sphere center with the sphere radius. Thus, an observation outside the sphere is classified as an outlier.

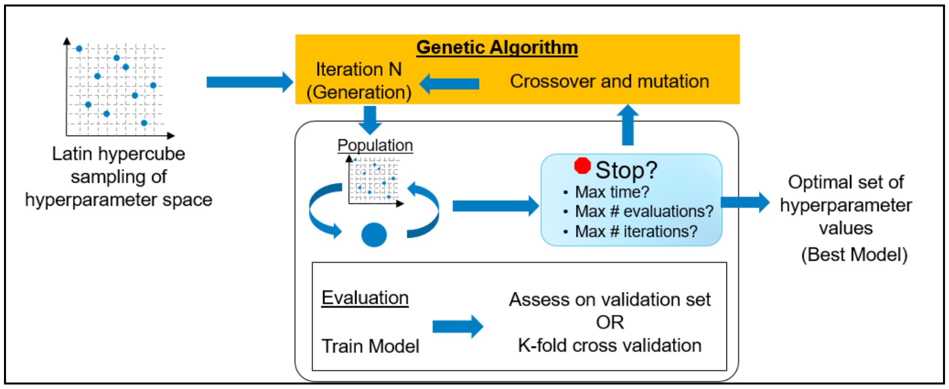


Figure 2. Autotuning Uses Optimization to Find the Best Set of Hyperparameters to Minimize Error

SAS VISUAL DATA MINING AND MACHINE LEARNING PRIMARY ANALYTIC INTERFACES

There are three primary interfaces we will cover in this paper. From within each tool, you can extend your analysis into one of the others. Data can be shared, and models can be extended and compared.

VISUAL ANALYTICS

SAS Visual Analytics enables drag-and-drop, exploratory visualization and modeling. Data must be loaded into memory, otherwise known as SAS Cloud Analytic Services (CAS). Once in CAS, you can interactively explore your data using visuals such as scatter plots, waterfall charts, bubble plots, time series plots and many more. As shown in Figure 3, you can further analyze your data using a set of statistics techniques including Clustering, Decision Trees, Generalized Linear Models, Linear Regression, and Logistic Regression. You can expand upon these models using the latest machine learning techniques including Factorization Machines, Forests, Gradient Boosting, Neural Networks, and Support Vector Machines.

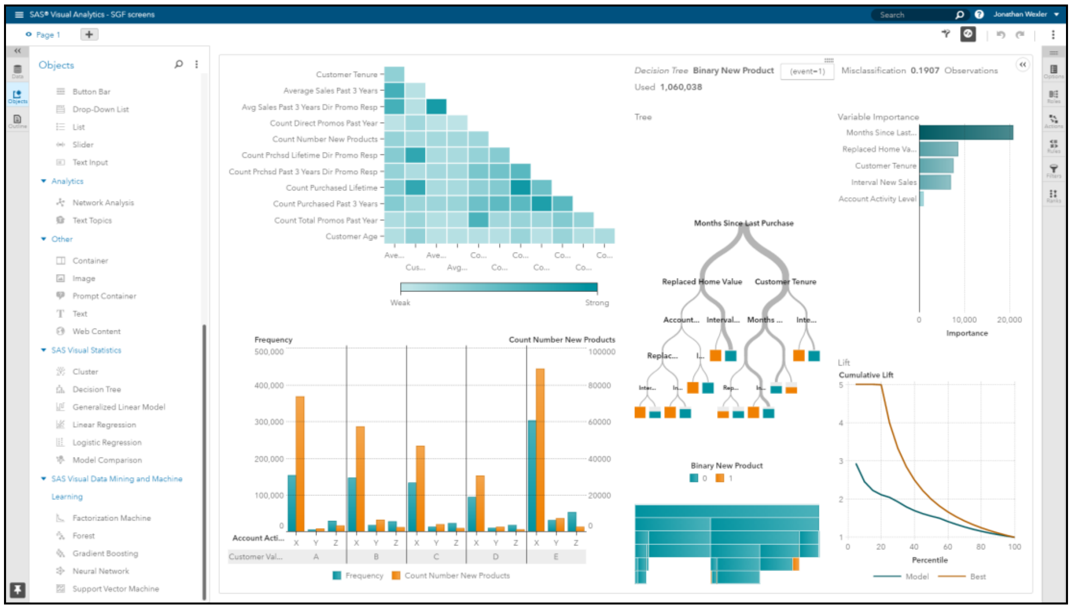
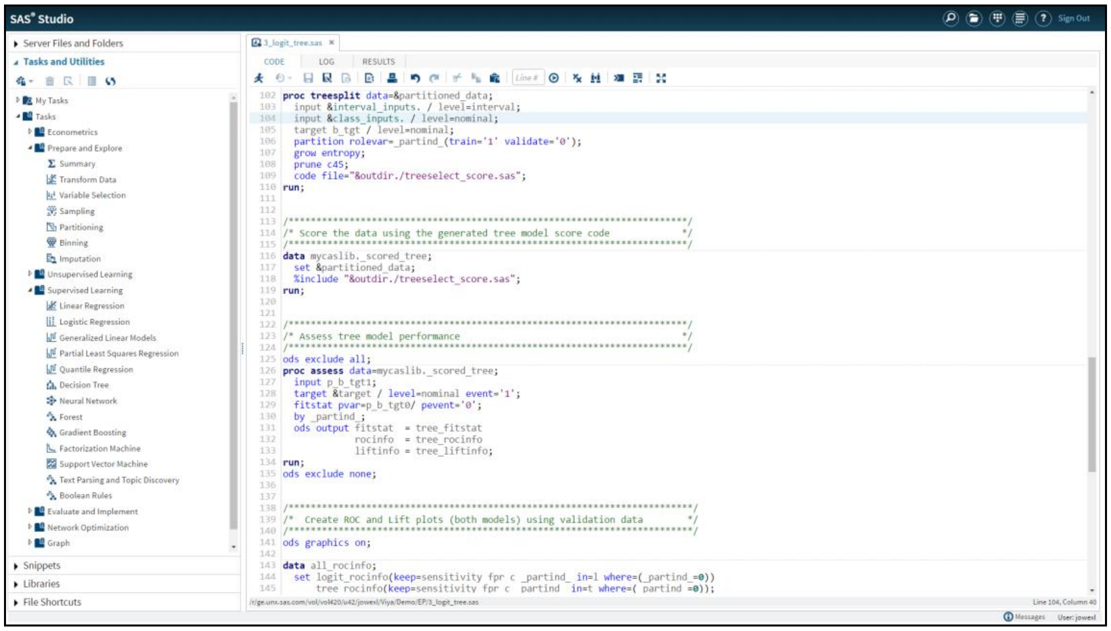


Figure 3. Interactive Visualization, Exploration, and Modeling Using SAS Visual Analytics

SAS STUDIO

SAS® Studio enables browser-based, programmatic access to the methods in SAS Viya. Using a modern, easy-to-use interface, you can run the exact same methods, and get the exact same answers as you would have with SAS Visual Analytics. As shown in Figure 4, you can programmatically run the methods from SAS Viya using in-memory procedures and SAS DATA step. Yes, the SAS DATA step now runs in-memory! There are several SAS Studio tasks that serve as code generators, so you have a way to learn and run these methods.



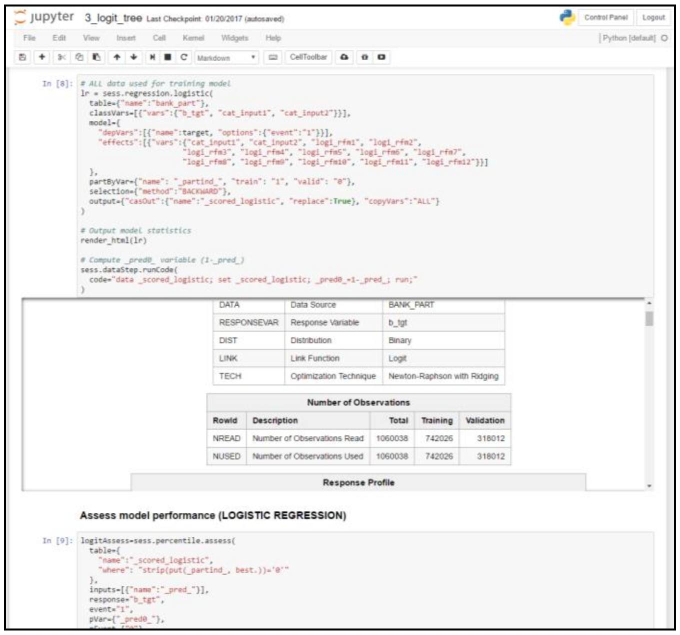
```
proc treesplit data=&partitioned_data;
  input &interval_inputs. / level=interval;
  input &class_inputs. / level=nominal;
  target b_tgt / level=nominal;
  partition rolevar=_partind_(train='1' validate='0');
  grow entropy;
  prune c45;
  code file="&outdir./treeselect_score.sas";
run;


/*************************************************************************/
/* Score the data using the generated tree model score code             */
/*************************************************************************/
data mycaslib._scored_tree;
  set &partitioned_data;
  %include "&outdir./treeselect_score.sas";
run;


/*************************************************************************/
/* Assess tree model performance                                         */
/*************************************************************************/
ods exclude all;
proc assess data=mycaslib._scored_tree;
  input p_b_tgt1;
  target &target / level=nominal event='1';
  fitstat pvar=p_b_tgt0/ pevent='0';
  by _partind_;
  ods output fitstat  = tree_fitstat
             rocinfo  = tree_rocinfo
             liftinfo = tree_liftinfo;
run;
ods exclude none;


/**************************************************************************/
/*  Create ROC and Lift plots (both models) using validation data         */
/**************************************************************************/
ods graphics on;

data all_rocinfo;
  set logit_rocinfo(keep=sensitivity fpr c _partind_ in=l where=(_partind_=0))
      tree_rocinfo(keep=sensitivity fpr c _partind_ in=t where=(_partind_=0));
```

/r/ge.unx.sas.com/vol/vol420/u42/jowexl/Viya/Demo/EP/3_logit_tree.sas

Figure 4. Access the SAS Viya Methods Using the SAS Language within SAS Studio

JUPYTER NOTEBOOK / PYTHON API

You can access the SAS Viya methods using the Python API to SAS Viya. The same methods that you can access in SAS Visual Analytics and SAS Studio are exposed from Python. A shown in Figure 5, you can access SAS Viya using a Jupyter notebook. Using a familiar Python construct, you can programmatically analyze your data, without any prior SAS knowledge.


jupyter 3_logit_tree Last Checkpoint: 01/20/2017 (autosaved)
Control Panel
Logout
File Edit View Insert Cell Kernel Widgets Help
Python [default]
Markdown
CellToolbar

```
In [8]: # ALL data used for training model
lr = sess.regression.logistic(
  table={"name":"bank_part"},
  classVars=[{"vars":{"b_tgt", "cat_input1", "cat_input2"}}],
  model={
    "depVars":[{"name":target, "options":{"event":"1"}}],
    "effects":[{"vars":{"cat_input1", "cat_input2", "logi_rfm1", "logi_rfm2",
                        "logi_rfm3", "logi_rfm4", "logi_rfm5", "logi_rfm6", "logi_rfm7",
                        "logi_rfm8", "logi_rfm9", "logi_rfm10", "logi_rfm11", "logi_rfm12"}}]
  },
  partByVar={"name": "_partind_", "train": "1", "valid": "0"},
  selection={"method":"BACKWARD"},
  output={"casOut":{"name":"_scored_logistic", "replace":True}, "copyVars":"ALL"}
)

# Output model statistics
render_html(lr)

# Compute _pred0_ variable (1-_pred_)
sess.dataStep.runCode(
  code="data _scored_logistic; set _scored_logistic; _pred0_=1-_pred_; run;"
)
```

DATA	Data Source	BANK_PART
RESPONSEVAR	Response Variable	b_tgt
DIST	Distribution	Binary
LINK	Link Function	Logit
TECH	Optimization Technique	Newton-Raphson with Ridging

Number of Observations				
RowId	Description	Total	Training	Validation
NREAD	Number of Observations Read	1060038	742026	318012
NUSED	Number of Observations Used	1060038	742026	318012

Response Profile

Assess model performance (LOGISTIC REGRESSION)

```
In [9]: logitAssess=sess.percentile.assess(
  table={
    "name":"_scored_logistic",
    "where": "strip(put(_partind_, best.))='0'"
  },
  inputs=[{"name":"_pred_"}],
  response="b_tgt",
  event="1",
  pVar={"_pred0_"},
```

Figure 5. Access the SAS Viya Methods Using the Python API to SAS Viya

CASE STUDY

The BANK data set contains more than one million observations (rows) and 24 variables (columns) for this case study. The data set comes from a large financial services firm and represents consumers' home equity lines of credit, their automobile loans, and other types of short to medium-term credit instruments. Note that the data has been anonymized and transformed to conform to the regulation guidelines.

Though three target variables are available in the data set, the primary focus is on the binary target variable B_TGT, which indicates consumer accounts that bought at least one product in the previous campaign season. A campaign season at the bank runs for half a year and encompasses all marketing efforts to motivate the purchase (contracting) of the bank's financial services products. Campaign promotions are categorized into direct and indirect -- direct promotions consist of sales offers to a particular account that involve an incentive while indirect promotions are marketing efforts that do not involve an incentive.

In addition to the account identifier (**Account ID**), the following tables describe the variables in the data set:

Name	Label	Description
B_TGT	Tgt Binary New Product	A binary target variable. Accounts coded with a 1 contracted for at least one product in the previous campaign season. Accounts coded with a 0 did not contract for a product in the previous campaign season.
INT_TGT	Tgt Interval New Sales	The amount of financial services product (sum of sales) per account in the previous campaign season, denominated in US dollars.
CNT_TGT	Tgt Count Number New Products	The number of financial services products (count) per account in the previous campaign season.

Table 2. Target Variables Quantify Account Responses over the Current Campaign Season.

Name	Label	Description
CAT_INPUT1	Category 1 Account Activity Level	A three-level categorical variable that codes the activity of each account. • X → high activity. The account enters the current campaign period with a lot of products. • Y → average activity. • Z → low activity.
CAT_INPUT2	Category 2 Customer Value Level	A five-level (A-E) categorical variable that codes customer value. For example, the most profitable and creditworthy customers are coded with an A.

Table 3. Categorical Inputs Summarize Account-level Attributes Related to the Propensity to Buy Products and Other Characteristics Related to Profitability and Creditworthiness. These Variables Have Been Transformed to Anonymize Account-level Information and to Mitigate Quality Issues Related to Excessive Cardinality.

Name	Label	Description
RFM1	RFM1 Average Sales Past 3 Years	Average sales amount attributed to each account over the past three years
RFM2	RFM2 Average Sales Lifetime	Average sales amount attributed to each account over the account's tenure
RFM3	RFM3 Avg Sales Past 3 Years Dir Promo Resp	Average sales amount attributed to each account in the past three years in response to a direct promotion
RFM4	RFM4 Last Product Purchase Amount	Amount of the last product purchased
RFM5	RFM5 Count Purchased Past 3 Years	Number of products purchased in the past three years
RFM6	RFM6 Count Purchased Lifetime	Total number of products purchased in each account's tenure.
RFM7	RFM7 Count Prchsd Past 3 Years Dir Promo Resp	Number of products purchased in the previous three years in response to a direct promotion
RFM8	RFM8 Count Prchsd Lifetime Dir Promo Resp	Total number of products purchased in the account's tenure in response to a direct promotion
RFM9	RFM9 Months Since Last Purchase	Months since the last product purchase
RFM10	RFM10 Count Total Promos Past Year	Number of total promotions received by each account in the past year
RFM11	RFM11 Count Direct Promos Past Year	Number of direct promotions received by each account in the past year
RFM12	RFM12 Customer Tenure	Customer tenure in months.

Table 4. Interval Inputs Provide Continuous Measures on Account-level Attributes Related to the

Recency, Frequency, and Sales Amounts (RFM). All Measures below Correspond to Activity Prior to the Current Campaign Season.

Name	Label	Description
DEMOG_AGE	Demog Customer Age	Average age in each account's demographic region
DEMOG_GENF	Demog Female Binary	A categorical variable that is 1 if the primary holder of the account if female and 0 otherwise.
DEMOG_GENM	Demog Male Binary	A categorical variable that is 1 if the primary holder of the account is male and 0 otherwise
DEMOG_HO	Demog Homeowner Binary	A categorical variable that is 1 if the primary holder of the account is a homeowner and 0 otherwise.
DEMOG_HOMEVAL	Demog Home Value	Average home value in each account's demographic region
DEMOG_INC	Demog Income	Average income in each account's demographic region
DEMOG_PR	Demog Percentage Retired	The percentage of retired people in each account's demographic region

Table 5. Demographic Variables Describe the Profile of Each Account in Terms of Income, Homeownership, and Other Characteristics.

LOAD LOCAL DATA TO IN-MEMORY LIBRARY

Before we start our analysis, we will use SAS Studio to load the local data to memory, so that it is accessible by our analytics team both visually and programmatically. We will then create a validation holdout set in order to assess our models.

The first LIBNAME statement automatically starts a CAS session, attached to the public caslib. Caslibs are in-memory locations that contain tables, access controls, and information about data sources. We are using the public caslib since this location is accessible by our team. In SAS Studio mycaslib is a library reference to the public caslib and will be referred to by SAS Viya procedures and any SAS DATA steps. The second LIBNAME statement is linked to the local file system that contains our SAS data set.

```
libname mycaslib cas caslib=public;
libname locallib 'your_local_library';
```

We will use PROC CASUTIL to load our local data to the public caslib. Using the 'promote' option enables us to make the data available to all CAS sessions. By default, tables in CAS sessions have local scope, so promoting enables you to access the in-memory table across multiple sessions and users.

```
proc casutil;
  load data=locallib.bank OUTCASLIB="public" casout="bank" promote;
run;
```

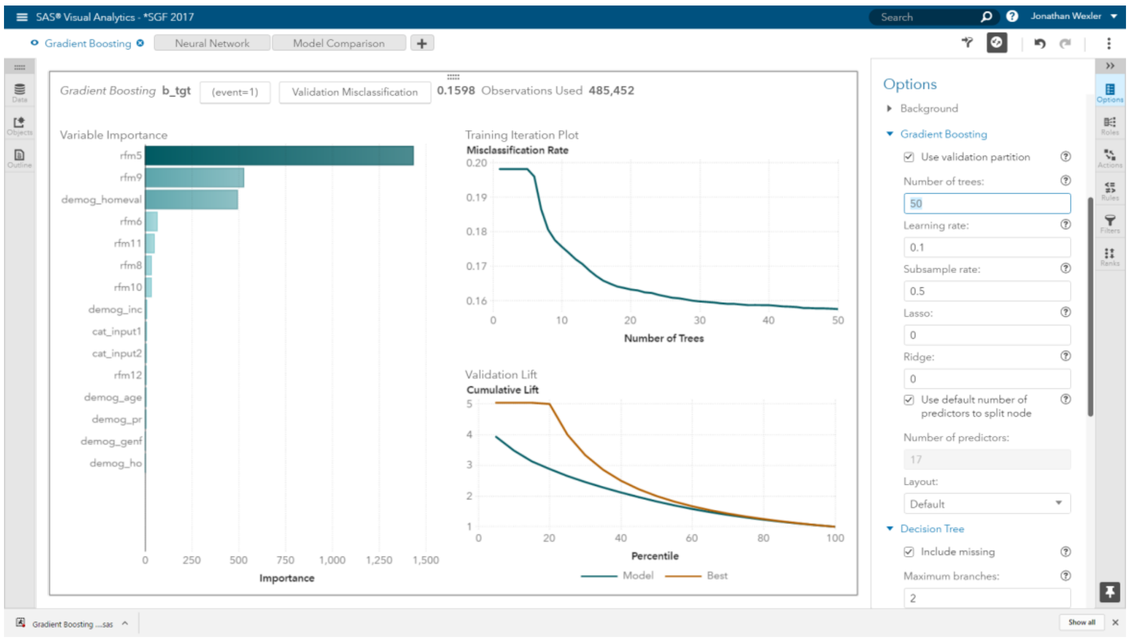
We will run PROC PARTITION to randomly separate our data into training and validation partitions. A new variable _partind_ will be assigned two numeric values: 1 for training data and 2 for validation data. The seed option allows you to re-create the random sample in future CAS sessions on the same CAS server. This is valuable when trying to reproduce results with multiple users. You should include the copyvars option if you want to keep all source variables in your partitioned data set.

```
proc partition data=mycaslib.bank partition samppct=70 seed=12345;
  by b_tgt;
  output out=mycaslib.bank_part copyvars=(_ALL_);
run;
```

BUILD MODELS INTERACTIVELY USING VISUAL ANALYTICS

Once the data is loaded and promoted to the public caslib, it is accessible from within SAS Visual Analytics. The first model we will create is a Gradient Boosting model, which trains a series of decision trees successively to fit the residual of the prediction from the earlier trees in the series. The target in this model is b_tgt. We will set the number of trees to 50. There are other regularization options such as Lasso and Ridge that can help prevent overfitting. As you change options, the visualization is recomputed in near real time, taking just a few seconds.

Note that misclassification for the validation partition is 0.1598. The first visualization, on the left, is the Variable Importance plot. This plot displays each variable's importance in the model. See that rfm5, rfm9, and demog_homeval are proportionately more important than the other predictors. The next plot is the Iteration plot, which indicates how well the model classified as the number of trees increased. In this case, the misclassification rate tails off after about 30 trees. The bottom right plot indicates how well the model assessed in terms of lift, misclassification, and ROC.



Gradient Boosting b_tgt (event=1) Validation Misclassification 0.1598 Observations Used 485,452

Options

Background

Gradient Boosting

☑ Use validation partition

Number of trees: 50

Learning rate: 0.1

Subsample rate: 0.5

Lasso: 0

Ridge: 0

☑ Use default number of predictors to split node

Number of predictors: 17

Layout: Default

Decision Tree

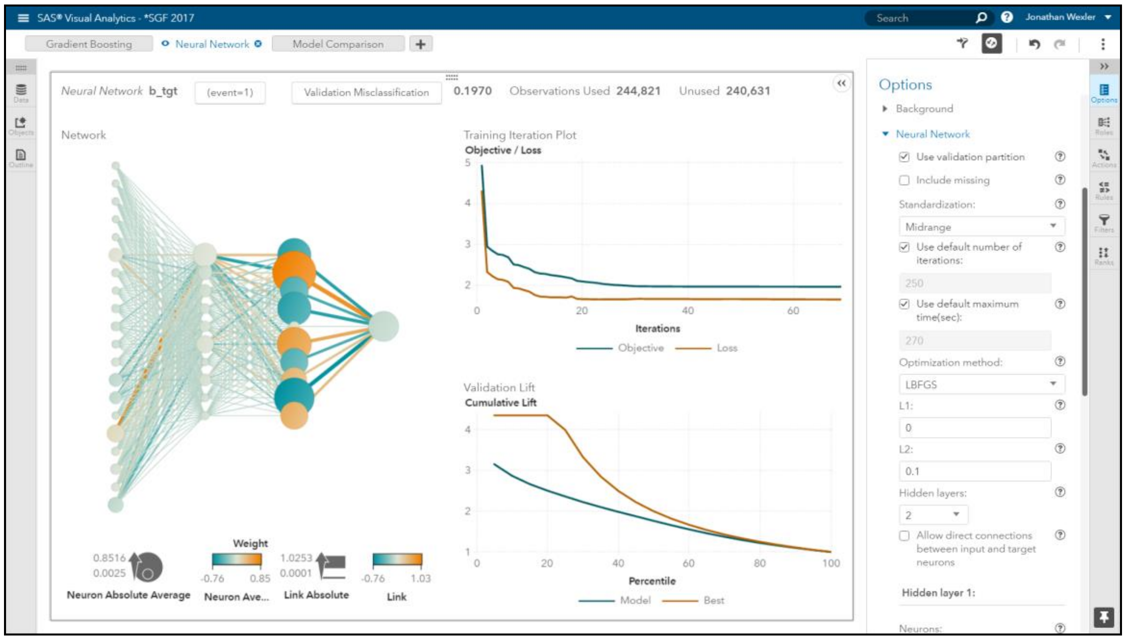
☑ Include missing

Maximum branches: 2

Figure 6. Interactive Gradient Boosting in SAS Visual Analytics

The next model we will build is a Neural Network, which is a statistical model that is designed to mimic the biological structures of the human brain that contains an input layer, multiple hidden layers, an output layer, and the connections between each of those.

Note that misclassification for the validation partition is 0.1970. The Network plot illustrates the relationship between your inputs and hidden layers. The next plot is the Iteration plot, which reports on the Objective/Loss function as the number of iterations increased. It appears that the Objective/Loss flattens around 20 iterations. You can tune the model further by changing the number of hidden layers, the number of neurons in each hidden layer, activation function for each layer, or other options.



SAS® Visual Analytics - *SGF 2017

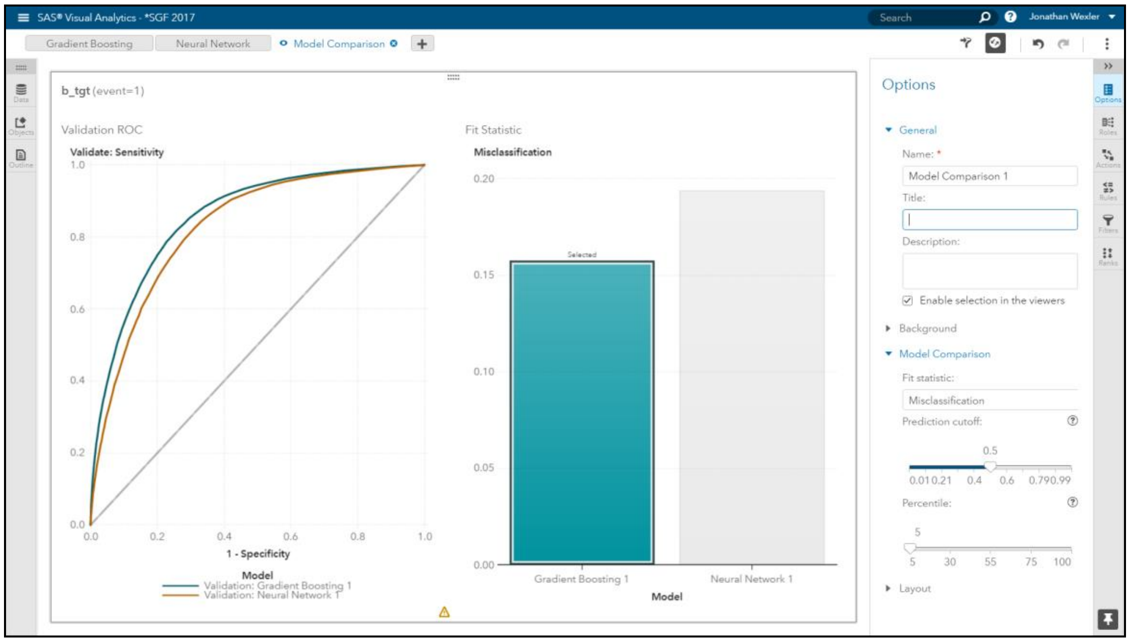
Neural Network b_tgt (event=1) Validation Misclassification 0.1970 Observations Used 244,821 Unused 240,631

Options

- Neural Network
  - ☑ Use validation partition
  - ☐ Include missing
  - Standardization: Midrange
  - ☑ Use default number of iterations: 250
  - ☑ Use default maximum time(sec): 270
  - Optimization method: LBFGS
  - L1: 0
  - L2: 0.1
  - Hidden layers: 2
  - ☐ Allow direct connections between input and target neurons
  - Hidden layer 1:
  - Neurons:

Figure 7. Interactive Two-Layer Neural Network in SAS Visual Analytics

The Model Comparison automatically chooses the best model based on the fit statistic selected in the Options panel. In this case, the model with the lowest misclassification rate is chosen. Note the partition, response, and event level much match across each model in order to generate the report. Gradient Boosting is selected as the champion model.



SAS® Visual Analytics - *SGF 2017

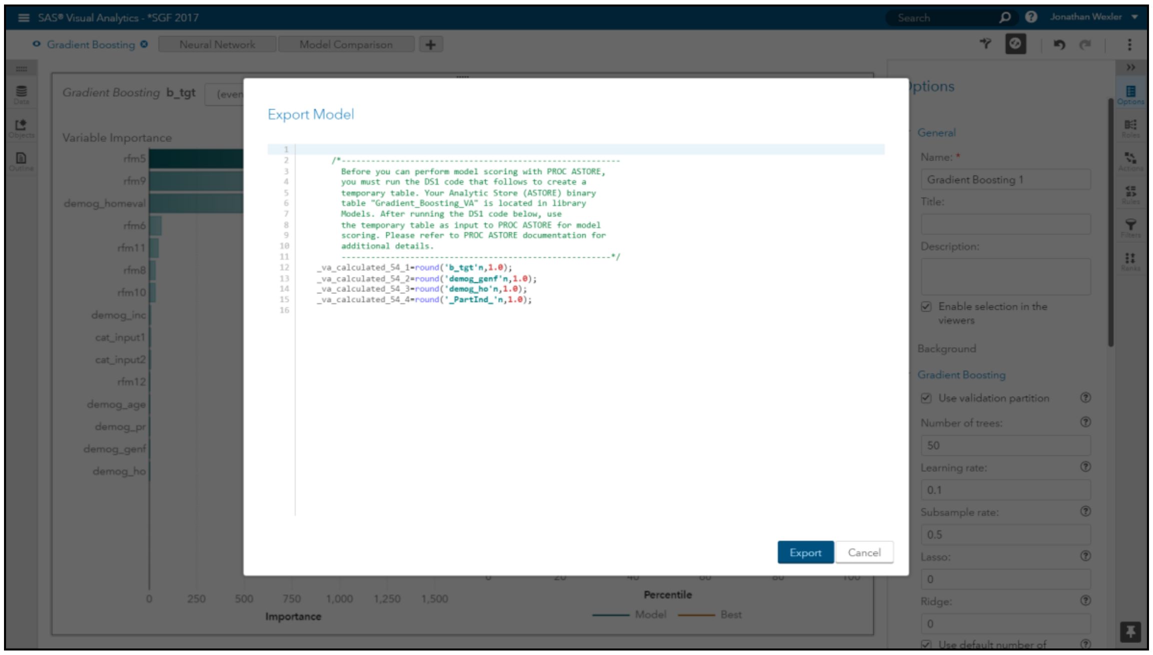
b_tgt (event=1)

Options

- General
  - Name: Model Comparison 1
  - Title:
  - Description:
  - ☑ Enable selection in the viewers
- Model Comparison
  - Fit statistic: Misclassification
  - Prediction cutoff: 0.5
  - Percentile: 5

Figure 8. Interactive Model Comparison in SAS Visual Analytics

We will export the Gradient Boosting model so that it is accessible from SAS Studio in the next section. This model information will automatically be stored in the 'models' caslib as a binary analytic store (or astore) file.



```
      /*-------------------------------------------------------
        Before you can perform model scoring with PROC ASTORE,
        you must run the DS1 code that follows to create a
        temporary table. Your Analytic Store (ASTORE) binary
        table "Gradient_Boosting_VA" is located in library
        Models. After running the DS1 code below, use
        the temporary table as input to PROC ASTORE for model
        scoring. Please refer to PROC ASTORE documentation for
        additional details.
        -------------------------------------------------------*/
    _va_calculated_54_1=round('b_tgt'n,1.0);
    _va_calculated_54_2=round('demog_genf'n,1.0);
    _va_calculated_54_3=round('demog_ho'n,1.0);
    _va_calculated_54_4=round('_PartInd_'n,1.0);
```

Figure 9. Exporting Gradient Boosting Score Code from SAS Visual Analytics

BUILD MODELS PROGRAMATICALLY USING SAS STUDIO

Now that we have explored our data and built interactive models within SAS Visual Analytics to predict b_tgt, we might want to extend our analysis and build additional models within SAS Studio, our programmatic environment. Prior to building our models within SAS Visual Analytics, we created and promoted the BANK_PART CAS table to the public caslib so that it is available across multiple sessions and multiple users. In addition, this table contains a _partind_ variable to represent our training and validation partitions. The LIBNAME statement below points to this public caslib and allows the user within SAS Studio to build models with the same table that was loaded into memory and used to build our interactive models in SAS Visual Analytics.

```
libname mycaslib cas caslib=public;
```

The first step in our modeling process is to further "wrangle" our data. In this case, we have identified several predictors that have a high percentage of missing values. In order to address this, we will first run PROC VARIMPUTE to replace these missing values with the calculated mean of all of the nonmissing observations.

```
%let partitioned_data = mycaslib.bank_part;

proc varimpute data=&partitioned_data.;
  input demog_age demog_homeval demog_inc rfm3  /ctech=mean;
  output out=mycaslib.bank_prepped_temp copyvars=(_ALL_);
  code file="&outdir./impute_score.sas";
run;
```

Next, we might want to apply transformations to a few of the continuous predictors. These transformations can be done using in-memory SAS DATA step code. Notice that the data being used to build these transformations as well as the output table that is being created are both pointing to a caslib. When this is the case, the SAS DATA step code is run automatically in-memory without requiring any special requests. This table is then promoted with the PROMOTE=YES option so that it can be used later if we want to continue the model building process with these new variables in an environment such as Python. We will show this type of integration in the next section.

```
%let prepped_data = mycaslib.bank_prepped;
data &prepped_data (promote=YES);
  set mycaslib.bank_prepped_temp ;

  if (IM_RFM3 > 0) then LOG_IM_RFM3 = LOG(IM_RFM3);
  else LOG_IM_RFM3 = .;

  if (RFM1 > 0) then LOG_RFM1 = LOG(RFM1);
  else LOG_RFM1 = .;
run;
```

The first model we will build is a Decision Tree model. Decision Trees use a sequence of simple if-then-else rules to make a prediction or to classify an output. We will build this model with PROC TREESPLIT using the Entropy growing criterion and then apply the C45 methodology to select the optimal tree, which is based on the validation partition. We store the details of this tree model in the score code file treeselect_score.sas. This score code is applied to the bank data creating new columns that contain the predicted value for each observation.

```
/* Specify the data set inputs and target */
%let class_inputs    = cat_input1 cat_input2 demog_ho demog_genf
                      demog_genm;
%let interval_inputs = IM_demog_age IM_demog_homeval IM_demog_inc
                       demog_pr log_rfm1 rfm2 log_im_rfm3 rfm4-rfm12 ;
%let target          = b_tgt;

/* DECISION TREE predictive model                                      */
proc treesplit data=&prepped_data.;
  input &interval_inputs. / level=interval;
  input &class_inputs. / level=nominal;
  target &target. / level=nominal;
  partition rolevar=_partind_(train='1' validate='0');
  grow entropy;
  prune c45;
  code file="&outdir./treeselect_score.sas";
run;

/* Score the data using the generated tree model score code            */
data mycaslib._scored_tree;
  set &prepped_data.;
  %include "&outdir./treeselect_score.sas";
run;
```

In Figure 10, we see a partial tree diagram that was created from running PROC TREESPLIT. This shows that the first rule applied to the data was based on the predictor rfm5. Those observations that have a value for rfm5 that was less than or equal to 3.6 were passed into the left hand branch; those with a value of rfm5 that was greater than 3.6 were passed into the right hand branch. You can continue to follow the rules down the entire branch of a tree until arriving at the final node, which determines your classification.

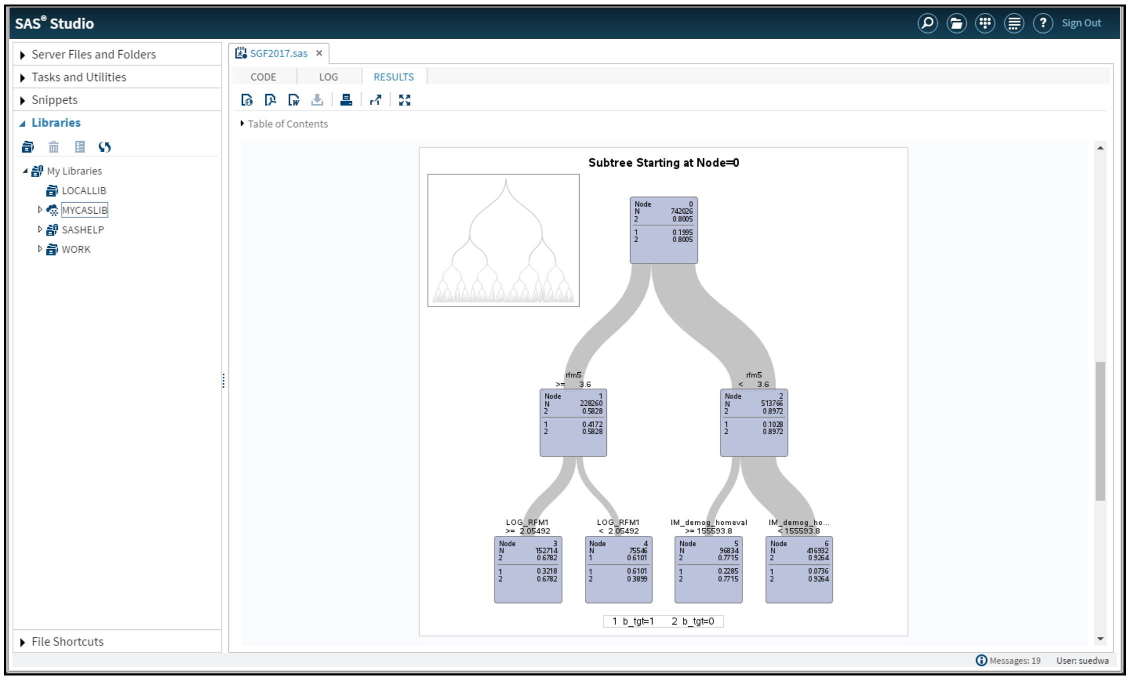
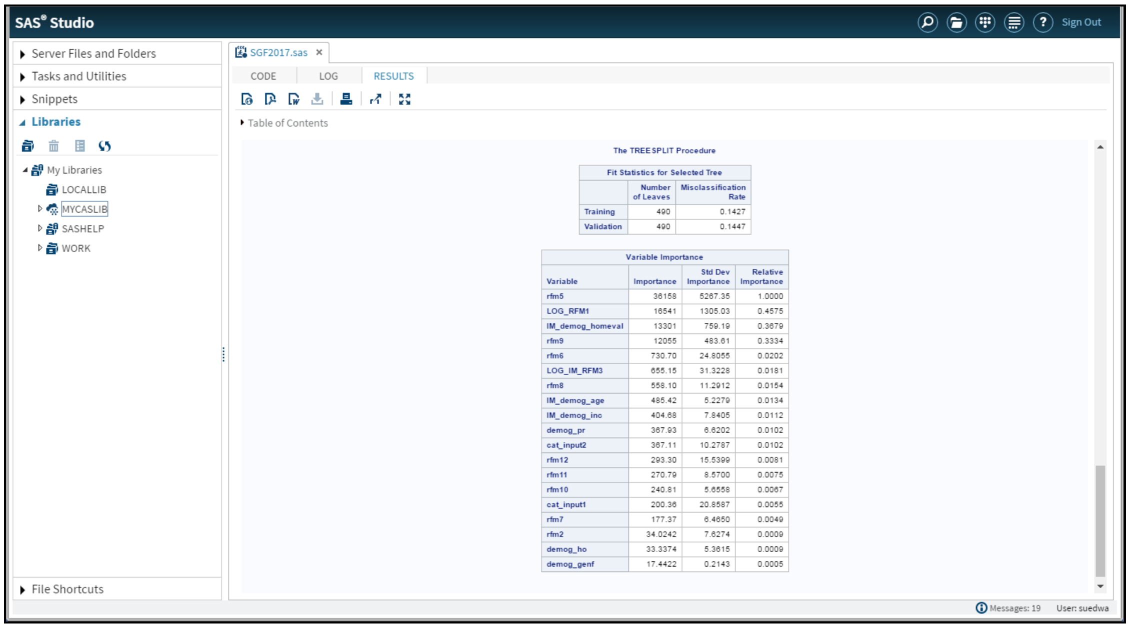


Figure 10. Decision Tree Subtree Diagram from SAS Studio

Note that the misclassification for the validation partition is 0.1447. The Variable Importance table reports the relative importance of all of the predictors that were used in building this model. We can see from this table that rfm5, LOG_RFM1, IM_demog_homeval, and rfm9 were the predictors that contributed the most in defining the splitting rules that made up this particular decision tree model.



The TREESPLIT Procedure

Fit Statistics for Selected Tree

	Number of Leaves	Misclassification Rate
Training	490	0.1427
Validation	490	0.1447

Variable Importance

Variable	Importance	Std Dev Importance	Relative Importance
rfm5	36158	5267.35	1.0000
LOG_RFM1	16541	1305.03	0.4575
IM_demog_homeval	13301	759.19	0.3679
rfm9	12055	483.61	0.3334
rfm6	730.70	24.8055	0.0202
LOG_IM_RFM3	655.15	31.3228	0.0181
rfm8	558.10	11.2912	0.0154
IM_demog_age	485.42	5.2279	0.0134
IM_demog_inc	404.68	7.8405	0.0112
demog_pr	367.93	6.6202	0.0102
cat_input2	367.11	10.2787	0.0102
rfm12	293.30	15.5399	0.0081
rfm11	270.79	8.5700	0.0075
rfm10	240.81	5.6558	0.0067
cat_input1	200.36	20.8587	0.0055
rfm7	177.37	6.4650	0.0049
rfm2	34.0242	7.6274	0.0009
demog_ho	33.3374	5.3615	0.0009
demog_genf	17.4422	0.2143	0.0005

Figure 11. Decision Tree Fit Statistics and Variable Importance Metrics from SAS Studio

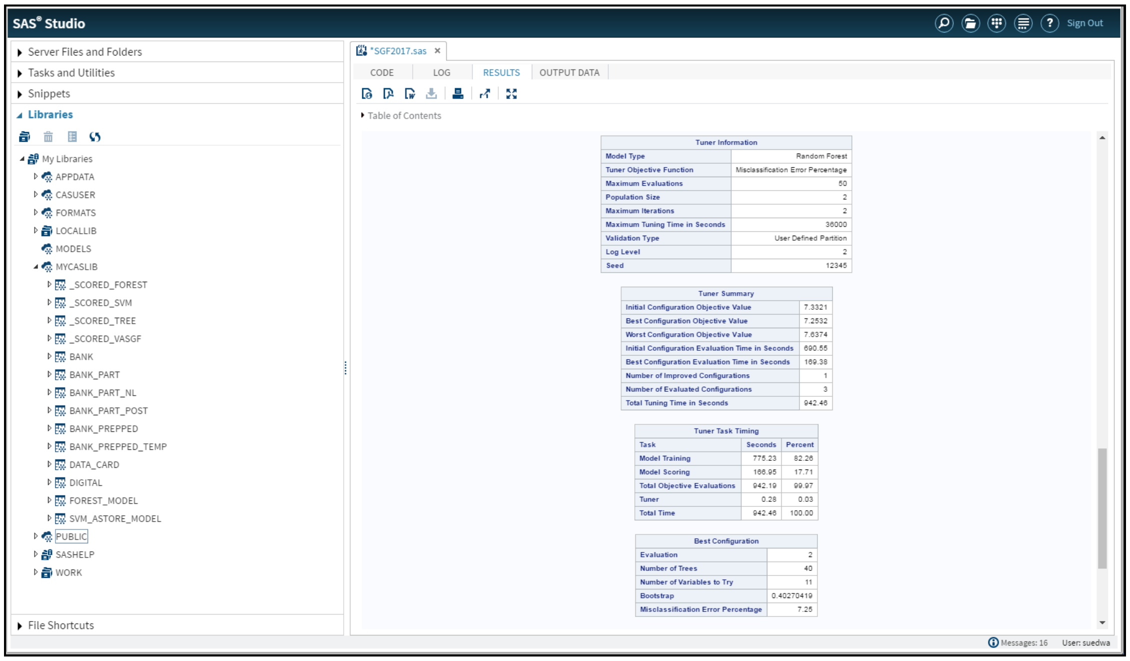
The next model that we will build is a Forest model. A Forest is an ensemble of individual trees where the final classification is based on an average of the probabilities across the trees that make up the forest. In many cases, finding the correct tuning parameters for a forest model can be quite tricky and time consuming. The autotuning options within PROC FOREST takes all of the guess work out of the tuning process and determines the optimal settings for these parameters based on the data. In this case, we allow PROC FOREST to autotune over the number of trees, the number of variables to try when splitting each node in the trees, and the in-bag fraction parameters for this model. We are using the outmodel option in this procedure to store all of the information about the model and to show an alternative to using score code. We can see mycaslib.forest_model being passed into the inmodel option within the second PROC FOREST call. This will be used to create our new output table containing our classifications for this model. Note that we could have specified the OUTPUT statement in the first PROC FOREST run because we are scoring the original input data. This approach would be used when you are scoring new data with the trained forest.

```
/* Autotune ntrees, vars_to_try and inbagfraction in Forest */
proc forest data=&prepped_data. intervalbins=20 minleafsize=5 seed=12345
outmodel=mycaslib.forest_model;
  input &interval_inputs. / level = interval;
  input &class_inputs. / level = nominal;
  target &target. / level=nominal;
  grow GAIN;
  partition rolevar=_partind_(train='1' validate='0');
  autotune maxiter=2 popsize=2 useparameters=custom
           tuneparms=(ntrees(lb=20 ub=100 init=100)
                      vars_to_try(init=5 lb=5 ub=20)
                      inbagfraction(init=0.6 lb=0.2 ub=0.9));
  ods output TunerResults=rf_tuner_results;
run;

/* Score the data using the generated Forest model */
proc forest data=&prepped_data. inmodel=mycaslib.forest_model noprint;
  output out=mycaslib._scored_FOREST copyvars=(b_tgt _partind_ account);
run;
```

In order to use this model in a different environment, such as in the next section where we use Python to build and compare models, the definition of this model must be promoted. For the Forest model, this definition was stored with the OUTMODEL option on the original PROC FOREST call creating the forest_model table. This promotion is done using PROC CASUTIL.

```
/* Promote the forest_model table */
proc casutil outcaslib="public" incaslib="public";
   promote casdata="forest_model";
quit;
```

Information about the autotuning process is shown in Figure 12. The Tuner Summary table details the optimization settings used to solve this problem. For example, the total tuning time of this model took 942.46 seconds with an Initial Objective Value of 7.3321 resulting in the Best Objective Value of 7.2532. The Best Configuration table shows that the optimal parameter settings for this data occurred at the second evaluation with 40 Trees, 11 Variables to Try, and a Bootstrap Sample (in-bag fraction) of 0.4027. Note that the misclassification for the validation partition is 0.0725.



Tuner Information	
Model Type	Random Forest
Tuner Objective Function	Misclassification Error Percentage
Maximum Evaluations	50
Population Size	2
Maximum Iterations	2
Maximum Tuning Time in Seconds	36000
Validation Type	User Defined Partition
Log Level	2
Seed	12345

Tuner Summary	
Initial Configuration Objective Value	7.3321
Best Configuration Objective Value	7.2532
Worst Configuration Objective Value	7.6374
Initial Configuration Evaluation Time in Seconds	690.55
Best Configuration Evaluation Time in Seconds	169.38
Number of Improved Configurations	1
Number of Evaluated Configurations	3
Total Tuning Time in Seconds	942.46

Tuner Task Timing		
Task	Seconds	Percent
Model Training	775.23	82.26
Model Scoring	166.95	17.71
Total Objective Evaluations	942.19	99.97
Tuner	0.28	0.03
Total Time	942.46	100.00

Best Configuration	
Evaluation	2
Number of Trees	40
Number of Variables to Try	11
Bootstrap	0.40270419
Misclassification Error Percentage	7.25

Figure 12. Forest Autotuning Metrics from SAS Studio

The final model that we will build is a Support Vector Machine. Support Vector Machines find a set of hyperplanes that best separate the levels of a binary target variable. We will use a polynomial kernel of degree 2 and store this complex model within a binary astore table called svm_astore_model. This table is then used in the PROC ASTORE call to generate the output table including your predicted classifications for this SVM model.

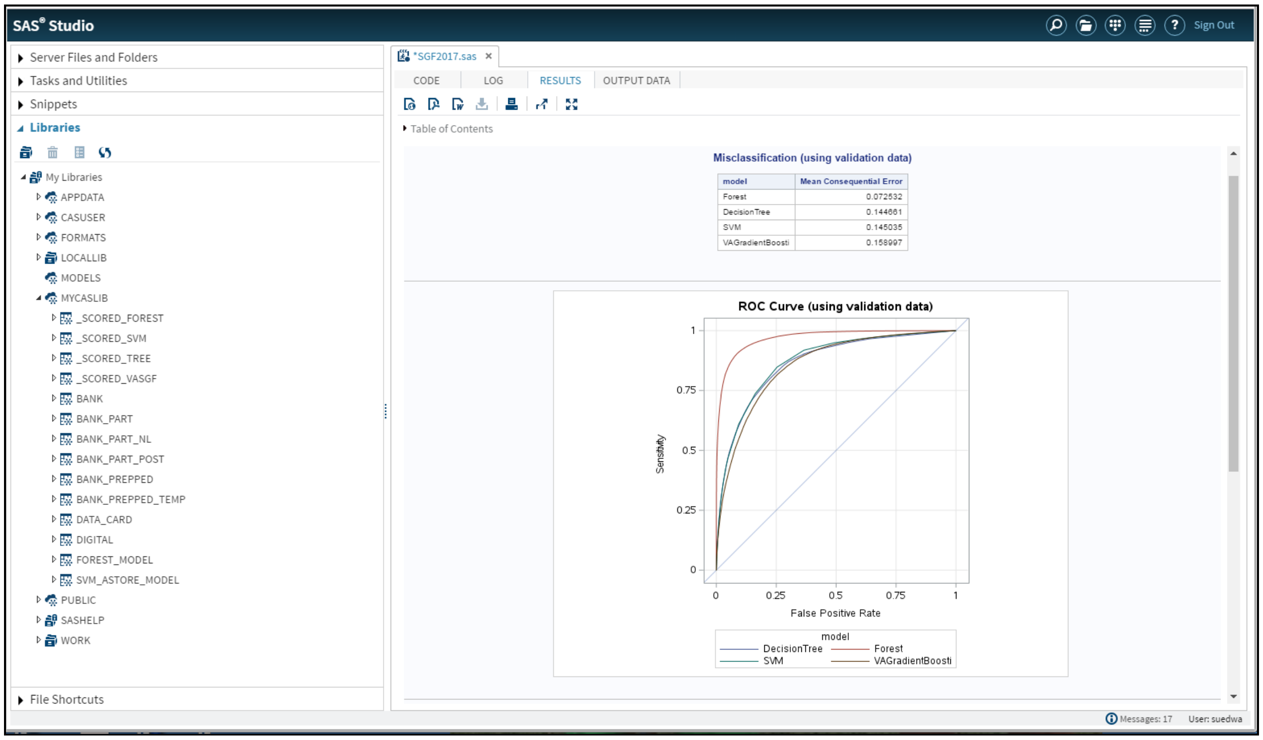
```
/* SUPPORT VECTOR MACHINE predictive model */
proc svmachine data=&prepped_data. (where=(_partind_=1));
  kernel polynom / deg=2;
  target &target. ;
  input &interval_inputs. / level=interval;
  input &class_inputs. / level=nominal;
  savestate rstore=mycaslib.svm_astore_model (promote=yes);
  ods exclude IterHistory;
run;

/* Score data using ASTORE code generated for the SVM model */
proc astore;
  score data=&prepped_data. out=mycaslib._scored_SVM
        rstore=mycaslib.svm_astore_model
        copyvars=(b_tgt _partind_  account);
run;

proc casutil outcaslib="public" incaslib="public";
   promote casdata="svm_astore_model";
quit;
```

Now that we have built several candidate models within SAS Studio, we want to compare these to each other to determine the best model for fitting this data. We also want to compare these with the original Gradient Boosting model, which was identified as the champion within SAS Visual Analytics. The details of this champion model were stored in an analytic store binary file and exported into the models library that is available within SAS Studio. To include this model in our comparisons, PROC ASTORE is run to apply this model to the BANK_PART data and to create the associated classifications.

```
proc casutil;
    Load casdata="Gradient_Boosting_VA.sashdat" incaslib="models"
    casout="gstate" outcaslib=casuser replace;
run;

data mycaslib.bank_part_post;
  set &partitioned_data.;
  _va_calculated_54_1=round('b_tgt'n,1.0);
  _va_calculated_54_2=round('demog_genf'n,1.0);
  _va_calculated_54_3=round('demog_ho'n,1.0);
  _va_calculated_54_4=round('_PartInd_'n,1.0);
run;

proc astore;
    score data=mycaslib.bank_part_post out=mycaslib._scored_vasgf
          rstore=casuser.gstate copyvars=(b_tgt _partind_ account ) ;
run;
```

These four candidate models are then passed to PROC ASSESS to calculate standard metrics including misclassification, lift, ROC, and more. Figure 13 shows that the best performing model for these candidates is the Forest model with a validation misclassification of 0.072532. This is also confirmed by looking at the ROC plot and the Lift values in the upper deciles.

```
/* Assess */
%macro assess_model(prefix=, var_evt=, var_nevt=);
  proc assess data=mycaslib._scored_&prefix.;
    input &var_evt.;
    target &target. / level=nominal event='1';
    fitstat pvar=&var_nevt. / pevent='0';
    by _partind_;

    ods output
      fitstat=&prefix._fitstat
      rocinfo=&prefix._rocinfo
      liftinfo=&prefix._liftinfo;
run;
%mend assess_model;

ods exclude all;
%assess_model(prefix=TREE, var_evt=p_b_tgt1, var_nevt=p_b_tgt0);
%assess_model(prefix=FOREST, var_evt=p_b_tgt1, var_nevt=p_b_tgt0);
%assess_model(prefix=SVM, var_evt=p_b_tgt1, var_nevt=p_b_tgt0);
%assess_model(prefix=VAGBM, var_evt=p_b_tgt1, var_nevt=p_b_tgt0);
ods exclude none;
```



Misclassification (using validation data)

model	Mean Consequential Error
Forest	0.072532
DecisionTree	0.144661
SVM	0.145035
VAGradientBoosti	0.158997

Figure 13. Assessment Statistics and ROC Curve for Candidate Models from SAS Studio

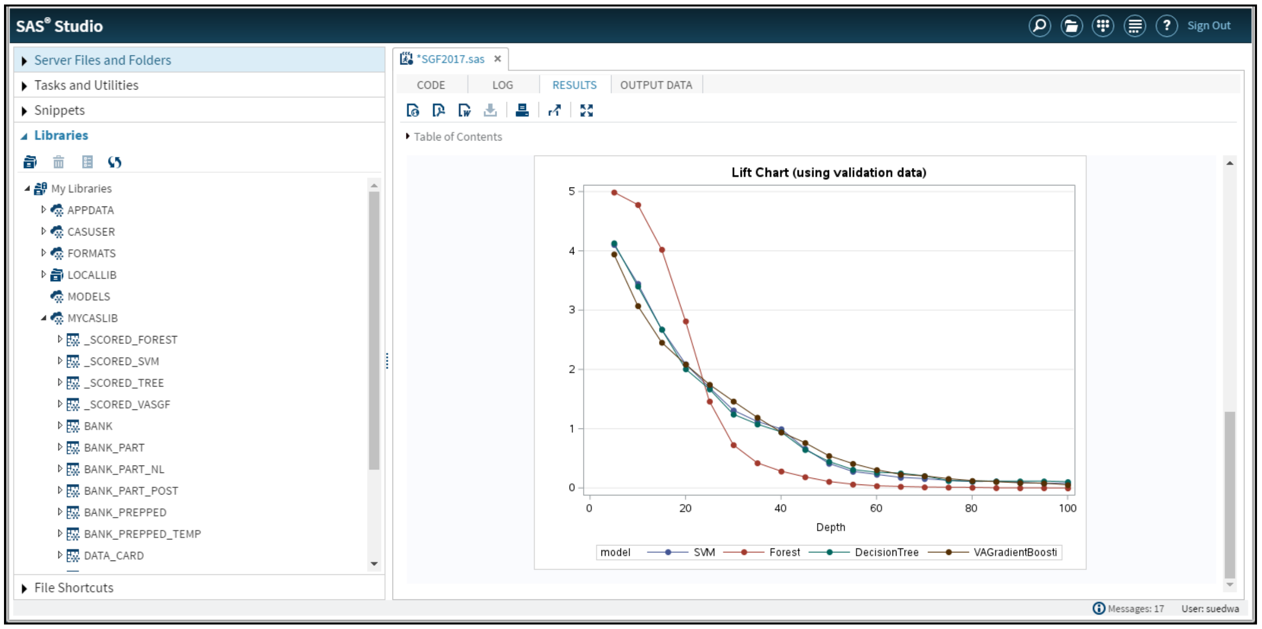


Figure 14. Lift Chart for Candidate Models from SAS Studio

BUILD MODEL PROGRAMMATICALLY USING PYTHON API

After exploring and modeling interactively in SAS Visual Analytics and programmatically in SAS Studio, we move into the open source world and finish this case study with another programmatic interface using the Python API. In this section, we will build a logistic regression model, score models that were built earlier in SAS Visual Analytics and SAS Studio, and compare them all to select a champion. The code and plots below are executed in Jupyter notebook.

We start by importing the SAS Scripting Wrapper for Analytics Transfer (SWAT) package to enable the connection and functionality of CAS. It is available at https://github.com/sassoftware/python-swat.

```
# Import packages
from swat import *
from pprint import pprint
from swat.render import render_html
from matplotlib import pyplot as plt
import pandas as pd
import sys
%matplotlib inline
```

The next step is to connect to CAS and start a new session. This step requires that you know the server host (cashost), port (casport), and authentication (casauth) of your CAS environment. Contact your SAS administrator for additional details and ensure that this code executes successfully before proceeding.

```
# Start a CAS session
cashost='cas_server_host.com'
casport=1234
casauth='~/_authinfo'
sess = CAS(cashost, casport, authinfo=casauth, caslib="public")
```

After execution, your CAS session can be accessed via the sess variable.

Next we define helper variables. Helper variables are those that are created in one place, at the beginning and reused afterward throughout the code. They include variables like the name of your input data set, its class and interval inputs, any shared caslibs, and so on.

```
# Set helper variables
gcaslib="public"
prepped_data="bank_prepped"
target = {"b_tgt"}
class_inputs = {"cat_input1", "cat_input2", "demog_ho", "demog_genf",
"demog_genm"}
interval_inputs = {"im_demog_age", "im_demog_homeval", "im_demog_inc",
"demog_pr", "log_rfm1", "rfm2", "log_im_rfm3", "rfm4", "rfm5", "rfm6",
"rfm7", "rfm8", "rfm9", "rfm10", "rfm11", "rfm12"}
class_vars = target | class_inputs
```

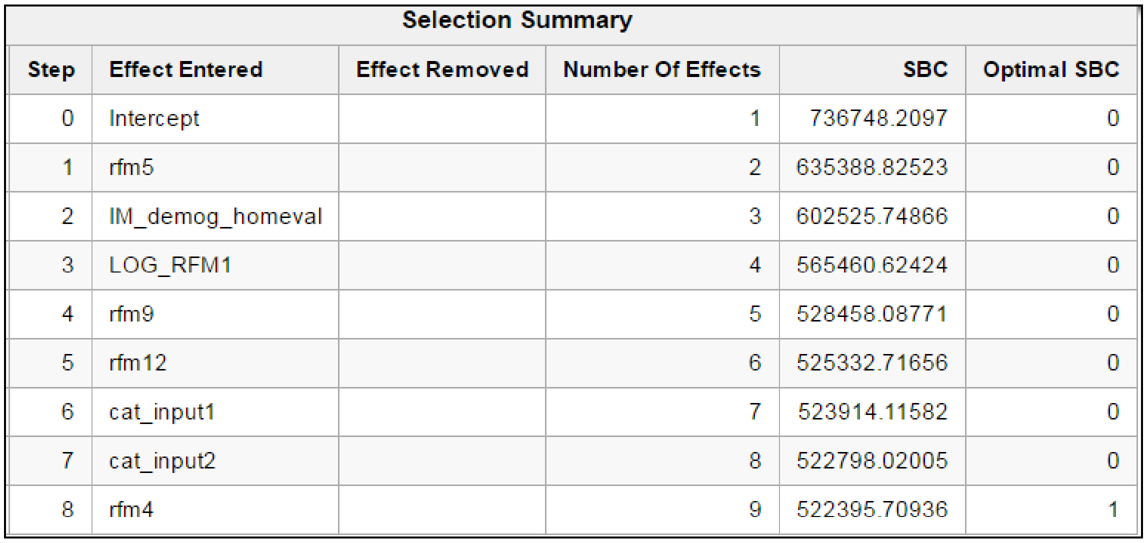
We begin by building a logistic regression model with stepwise selection, using the same set of inputs and target (b_tgt) used in the SAS Studio interface. Logistic regression models a binary target (0 or 1) and computes probabilities of the target event (1) as a function of specified inputs. This model uses the training partition of BANK_PREPPED table that was created and promoted to public caslib in SAS Studio. Because the table is promoted, it is available to any session on CAS, including ours.

After the model is run, the parameter estimates, fit statistics, and so on are displayed using render_html function from swat.render package. The Selection Summary in Figure 15 below lists the order of input variables selected at each step based on the SBC criterion. The misclassification rate for the validation partition is 0.1569. Finally, the predicted probabilities p_b_tgt0 and p_b_tgt1 are created using SAS DATA step code through the dataStep.runCode CAS action – these are needed later when invoking the model assessment function asses_model.

Note: Before invoking any CAS action, make sure the appropriate CAS actionset is loaded using sess.loadactionset. In the code below, notice that the regression actionset is loaded before the logistic action is invoked.

```
# Load action set
sess.loadactionset(actionset="regression")
```

```
# Train Logistic Regression
lr=sess.regression.logistic(
  table={"name":prepped_data, "caslib":gcaslib},
  classVars=[{"vars":class_vars}],
  model={
    "depVars":[{"name":"b_tgt", "options":{"event":"1"}}],
    "effects":[{"vars":class_inputs | interval_inputs}]
  },
  partByVar={"name":"_partind_", "train":"1", "valid":"0"},
  selection={"method":"STEPWISE"},
  output={"casOut":{"name":"_scored_logistic", "replace":True},
"copyVars":{"account", "b_tgt", "_partind_"}}
)

# Output model statistics
render_html(lr)

# Compute p_b_tgt0 and p_b_tgt1 for assessment
sess.dataStep.runCode(
  code="data _scored_logistic; set _scored_logistic; p_b_tgt0=1-_pred_;
rename _pred_=p_b_tgt1; run;"
)
```



Selection Summary

Step	Effect Entered	Effect Removed	Number Of Effects	SBC	Optimal SBC
0	Intercept		1	736748.2097	0
1	rfm5		2	635388.82523	0
2	IM_demog_homeval		3	602525.74866	0
3	LOG_RFM1		4	565460.62424	0
4	rfm9		5	528458.08771	0
5	rfm12		6	525332.71656	0
6	cat_input1		7	523914.11582	0
7	cat_input2		8	522798.02005	0
8	rfm4		9	522395.70936	1

Figure 15. Selection Summary of Logistic Regression Model from Python API

After building a model using the Python API, let us score few models created in SAS Visual Analytics and SAS Studio to understand how a model created in one interface can be shared and reused in another. We will begin with the Gradient Boosting model created in SAS Visual DATA steps. When this model was built, it produced two artifacts: SAS data step code and an astore file that was saved to models caslib.

To score the Gradient Boosting model using these artifacts, the code does the following:

1. Loads the astore file into a local user caslib (casuser)
2. Runs SAS DATA step code created in SAS Visual Analytics – this transforms the input data set BANK_PREPPED with any necessary changes made within this interface
3. Scores the transformed input data set (from step 2) using the loaded astore file (from step 1) that contains model parameters
4. Renames predicted probability variable names for assessment

```
# 1. Load GBM model (ASTORE) created in VA
sess.loadTable(
  caslib="models", path="Gradient_Boosting_VA.sashdat",
  casout={"name":"gbm_astore_model","caslib":"casuser", "replace":True}
)

# 2. Score code from VA (for data preparation)
sess.dataStep.runCode(
  code="""data bank_part_post;
             set bank_part(caslib='public');
             _va_calculated_54_1=round('b_tgt'n,1.0);
             _va_calculated_54_2=round('demog_genf'n,1.0);
             _va_calculated_54_3=round('demog_ho'n,1.0);
             _va_calculated_54_4=round('_PartInd_'n,1.0);
           run;"""
)

# 3. Score using ASTORE
sess.loadactionset(actionset="astore")

sess.astore.score(
  table={"name":"bank_part_post"},
  rstore={"name":"gbm_astore_model"},
  out={"name":"_scored_gbm", "replace":True},
  copyVars={"account", "_partind_", "b_tgt"}
)

# 4. Rename p_b_tgt0 and p_b_tgt1 for assessment
sess.dataStep.runCode(
  code="""data _scored_gbm;
             set _scored_gbm;
             rename p__va_calculated_54_10=p_b_tgt0
                    p__va_calculated_54_11=p_b_tgt1;
           run;"""
)
```

We repeat the scoring process with the autotuned Forest model created in SAS Studio. Remember that this model was saved earlier as a CAS table called forest_model in the public caslib. Here the decisionTree.forestScore action scores the input data set BANK_PREPPED using the forest_model table. The SAS DATA step that follows creates the necessary predicted probability variable names for assessment.

```
# Load action set
sess.loadactionset(actionset="decisionTree")

# Score using forest_model table
sess.decisionTree.forestScore(
  table={"name":prepped_data, "caslib":gcaslib},
  modelTable={"name":"forest_model", "caslib":"public"},
  casOut={"name":"_scored_rf", "replace":True},
  copyVars={"account", "b_tgt", "_partind_"},
  vote="PROB"
)
```

```
# Create p_b_tgt0 and p_b_tgt1 as _rf_predp_ is the probability of event in
_rf_predname_
sess.dataStep.runCode(
  code="""data _scored_rf;
            set _scored_rf;
            if _rf_predname_=1 then do;
              p_b_tgt1=_rf_predp_;
              p_b_tgt0=1-p_b_tgt1;
            end;
            if _rf_predname_=0 then do;
              p_b_tgt0=_rf_predp_;
              p_b_tgt1=1-p_b_tgt0;
            end;
          run;"""
)
```

Lastly we score the Support Vector Machine model created in SAS Studio using the analytic store (astore) table svm_astore_model located in public caslib.

```
# Score using ASTORE
sess.loadactionset(actionset="astore")

sess.astore.score(
  table={"name":prepped_data, "caslib":gcaslib},
  rstore={"name":"svm_astore_model", "caslib":"public"},
  out={"name":"_scored_svm", "replace":True},
  copyVars={"account", "_partind_", "b_tgt"}
)
```

The final step in the case study is to assess and compare all of the models that were created and scored, including both the interactively and programmatically created models. The assessment is based on the validation partition of the data. The code below uses the percentile.assess action for Logistic Regression model but similar code can be used to generate assessments for all other models.

```
# Assess models
def assess_model(prefix):
    return sess.percentile.assess(
      table={
        "name":"_scored_" + prefix,
        "where": "strip(put(_partind_, best.))='0'"
      },
      inputs=[{"name":"p_b_tgt1"}],
      response="b_tgt",
      event="1",
      pVar={"p_b_tgt0"},
      pEvent={"0"}
    )

lrAssess=assess_model(prefix="logistic")
lr_fitstat =lrAssess.FitStat
lr_rocinfo =lrAssess.ROCInfo
lr_liftinfo=lrAssess.LIFTInfo
```

To choose a champion, we will use the ROC and Lift plots. Figures 16 and 17 shows that the autotuned Forest (SAS Studio) is the winner compared to the Logistic Regression (Python API), Support Vector

Machine (SAS Studio) and Gradient Boosting (SAS Visual Analytics) models as it has higher lift and more area under the ROC curve.

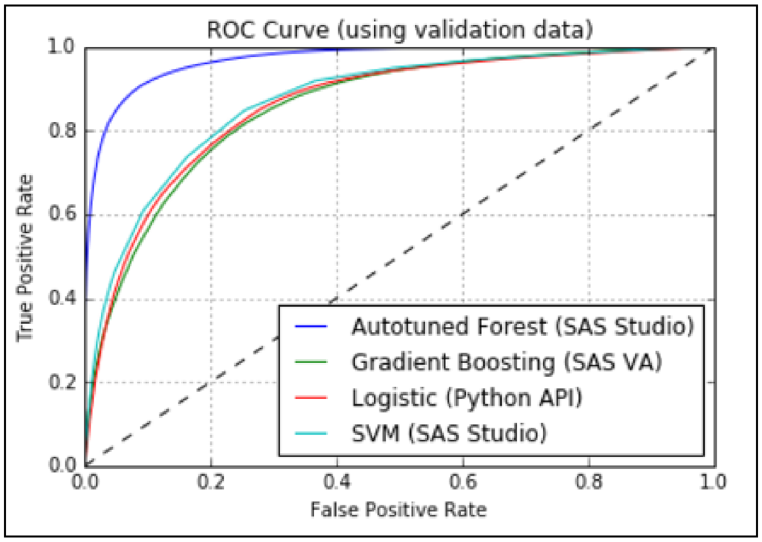


Figure 16. ROC Chart for Candidate Models

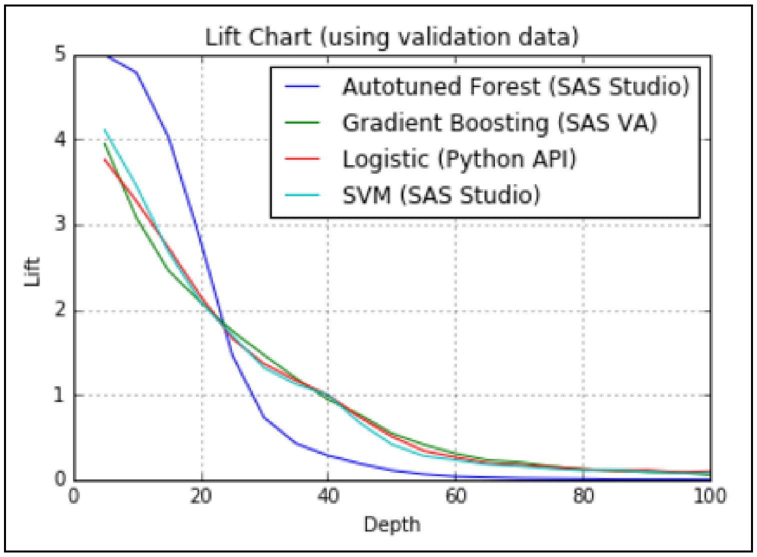


Figure 17. Lift Chart for Candidate Models

The goal of this case study is to highlight the unified and open architecture of SAS Viya -- how models built across various interfaces (SAS Visual Analytics, SAS Studio, and Python API) can seamlessly access data sets and intermediary results and easily score across them. Now that you understand the basics, you can build the best predictive model possible.

CONCLUSION

As previously stated, you should be able to solve business problems using your tool *and* method of choice, with no technological limitations. As shown in this paper, you can interactively build models quickly and accurately, and continue your analysis programmatically, without sacrificing inaccuracy from inefficient manual handoffs.

SAS Viya enables you to explore your data deeper, using the latest innovations in in-memory analytics. SAS is committed to delivering new, innovative data mining and machine learning algorithms that will scale to the size of your business, now and in the future.

REFERENCES

Koch, P., Wujek, B., Golovidov, O., and Gardner, S. (2017). "Automated Hyperparameter Tuning for Effective Machine Learning." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The authors express sincere gratitude to the SAS® Visual Data Mining and Machine Learning developers, testers, and also to our customers.

RECOMMENDED READING AND ASSETS

- SAS Visual Analytics, SAS Visual Statistics, and SAS Visual Data Mining and Machine Learning 8.1 on SAS Viya: Video Library (Visual)
  http://support.sas.com/training/tutorial/viyava/

- SAS Visual Data Mining and Machine Learning on SAS Viya: Video Library (Programming)
  http://support.sas.com/training/tutorial/viya/index.html

- SAS Visual Data Mining and Machine Learning Fact Sheet
  http://www.sas.com/content/dam/SAS/en_us/doc/factsheet/sas-visual-data-mining-machine-learning-1082751.pdf

- SAS Visual Data Mining and Machine Learning Community
  https://communities.sas.com/t5/SAS-Visual-Data-Mining-and/bd-p/dmml

- SAS Viya Documentation
  http://support.sas.com/documentation/onlinedoc/viya/

- SAS Software Github Page
  https://github.com/sassoftware

CONTACT INFORMATION

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

Jonathan Wexler
SAS Institute Inc.
100 SAS Campus Drive
Cary, NC 27513
Email: jonathan.wexler@sas.com

Susan Haller
SAS Institute Inc.
100 SAS Campus Drive
Cary, NC 27513
Email: susan.haller@sas.com

Radhikha Myneni
SAS Institute Inc.
100 SAS Campus Drive
Cary, NC 27513
Email: radhikha.myneni@sas.com