Demo Steps: Using the Python API to Create, Score, and Assess Models

From Jupyter Lab, select **File Browser** > **Home** > **Courses** > **EVMLOPRC** > **Notebooks** and select the **Python_Machine_Learning_Demo.ipynb** notebook. Scroll to Part 2.

1. Partition the **HMEQ** data set into 70% for training and 30% for validation.

```
conn.sampling.srs(
    table   = indata,
    samppct = 70,
    seed = 919,
    partind = True,
    output  = dict(casOut = dict(name = indata, replace = True),
                   copyVars = 'ALL')
)
```

   Selected arguments:

Argument	Description
table	specifies the in-memory input data table. The table is used in combination with the CAS session connection object.
samppct	specifies the sample percentage to be used for sampling or partitioning.
seed	specifies the integer to use to start the pseudorandom number generator.
partind	when set to *true*, generates a partition indicator column in the output table.
output	creates, on the server, a table that contains the sample output or partition output.
casOut	specifies the settings for an output table.
name	specifies the in-memory table name to save the action results.
replace	when set to ***true***, replaces the table with the results of the action.
copyVars	specifies a list of one or more variables to be copied from the input table to the output table. The keyword ALL copies all variables.

   The action added only a single binary partition indicator variable to the data set by using the partind=TRUE argument (partind = 1 for training and partind = 0 for validation).

2. Use SQL to find the count and percent of each level of the new **_PartInd_** variable.

```
conn.loadActionSet('fedSql')
actions = conn.builtins.help(actionSet='fedSql')

counts = conn.fedSql.execDirect(query =
    '''
    SELECT _PartInd_, count(*)
    FROM hmeq
    GROUP BY _PartInd_;
    '''
)['Result Set']

display(counts)
counts['Percent'] = counts['COUNT']/sum(counts['COUNT'])
counts
```

   The **counts** table is copied and brought to the client. Native open source syntax can act on the table further. In general, SQL can be used equivalently to PROC SQL. Under the fedSql action set and execDirect action, SQL takes advantage of the distributed environment and parallel processing.

```
counts$Percent = counts$COUNT/sum(counts$COUNT)
counts
```

   The result from the query is a data frame. Open source code is then used to add the **percent** column to the table. The percent is equivalent to the sampling proportion from the srs action.

3. Using the **defCasTable** function, refresh the object reference to the **HMEQ** data set and then use the SWAT function mean to find the proportion of training cases specified by the **_PartInd_** variable.

```
castbl = conn.CASTable(name=indata)
castbl['_PartInd_'].mean()
```

4. Use the super action from the varReduce action set to perform supervised dimension reduction. The action identifies a set of variables that jointly explain the maximum amount of variance that is contained in the response variables. Find the number of variables that are needed to explain 90% of the variability in the response.

```
conn.loadActionSet('varReduce')
actions = conn.builtins.help(actionSet='varReduce')

varReduce_obj = conn.varReduce.super(
    table = indata,
    target = target,
    inputs = inputs,
    nominals = nominals,
    varexp = 0.90
)

list(varReduce_obj)
```

   Selected arguments:

Argument	Description

table	specifies the in-memory input data table. The table is used in combination with the CAS session connection object.
target	specifies the target variable to use for analysis.
inputs	specifies the input variables to use for analysis.
nominals	specifies the nominal variables to use for analysis.
varexp	specifies the fraction of the total variance to be explained.

```
varReduce_obj['SelectionSummary']

display(varReduce_obj['SelectedEffects'])
list(varReduce_obj['SelectedEffects']['Variable'])
```

The super action selected all 12 inputs to explain 90% of the variability in the target. For larger data sets, it can be helpful to find a subset of inputs before you begin the modeling phase of the analysis.

The super action selects variables by using the discriminant criterion that is specified in linear discrimination analysis.

5. Use the logistic action from the regression action set to train a logistic regression model.

```
conn.loadActionSet('regression')
actions = conn.builtins.help(actionSet='regression')

conn.regression.logistic(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    classVars = nominals[1:],
    model = dict(depvar=target, effects=inputs, dist='binomial',
                 link='logit'),
    store    = dict(name='lr_model',replace=True)
)
```

Selected arguments:

Argument	Description
table	specifies the in-memory input data table. The table is used in combination with the CAS session connection object.
where	specifies a conditional argument for the observations to be used in the analysis.
classVars	names the classification variables for the analysis.
model	names the dependent variable, explanatory variable, and model options.
store	saves the regression model.

Note: The logistic action can also perform regularization and stepwise selection.

6. Use the svmTrain action from the svm action set to train a support vector machine.

```
conn.loadActionSet('svm')
actions = conn.builtins.help(actionSet='svm')

conn.svm.svmTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    target   = target,
    inputs   = inputs,
    nominals = nominals,
    kernel = 'polynomial',
    degree = 2,
    savestate = dict(name = 'svm_model', replace = True)
)
```

Selected arguments:

Argument	Description
kernel	specifies the kernel type.
Degree	specifies the degree of the polynomial kernel.
savestate	specifies the table to save the model for future scoring.

7. Load the decisionTree action set. Notice that it provides all the actions to train and score all three tree-based models. Use the dtreeTrain action to train a decision tree.

```
conn.loadActionSet('decisionTree')
actions = conn.builtins.help(actionSet='decisionTree')

conn.decisionTree.dtreeTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    target   = target,
    inputs   = inputs,
    nominals = nominals,
    casOut   = dict(name = 'dt_model', replace = True)
)
```

Selected arguments:

Argument	Description
table	specifies the in-memory input data table.
target	specifies the target variable to use for analysis.
inputs	specifies the input variables to use for analysis.
nominals	specifies the nominal variables to use for analysis.
casOut	specifies the table to store the decision tree model. If it is not specified, a random name is generated.

The model information displays the default parameters for the decision tree model. You can use the arguments within the action to change them.

8. Use the forestTrain action to train a random forest and use 1000 trees in the forest.

```
conn.decisionTree.forestTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    target   = target,
    inputs   = inputs,
    nominals = nominals,
    nTree    = 1000,
    casOut   = dict(name = 'rf_model', replace = True)
)
```

Selected argument:

Argument	Description
nTree	specifies the number of trees to create.

9. Use the gbtreeTrain action to train a gradient-boosting model and use 1000 tree iterations.

```
conn.decisionTree.gbtreeTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    target   = target,
    inputs   = inputs,
    nominals = nominals,
    nTree    = 1000,
    casOut   = dict(name = 'gbt_model', replace = True)
)
```

10. Load the neuralNet action set and use the annTrain action to train a neural network. Use a single hidden layer with 150 neurons.

```
conn.loadActionSet('neuralNet')
actions = conn.builtins.help(actionSet='neuralNet')

conn.neuralNet.annTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    target   = target,
    inputs   = inputs,
    nominals = nominals,
    hiddens  = [150],
    nloOpts  = dict(optmlOpt = dict(maxIters = 100,
                    fConv = 1e-10),
                    lbfgsOpt = dict(numCorrections = 6)),
    casOut   = dict(name = 'nn_model', replace = True)
)
```

Selected arguments:

Argument	Description
hiddens	specifies the number of neurons for each hidden layer. For example, hiddens={100,50} specifies two hidden layers, one with 100 neurons and the other with 50.
nloOpts	specifies the nonlinear optimization options.
maxIters	specifies the maximum iterations allowed for optimization. The default is 10.
fConv	specifies a stopping value when the objective functions fail to change more than a value in the allotted iterations. The default is 1e-05.
lbfgsOpt	specifies options for the Broyden-Fletcher-Goldfarb_shanno algorithm, an iterative method for solving unconstrained nonlinear optimization problems.
numCorrections	specifies the number of corrections used in the LBFGS update. The default is 20.
casOut	specifies the in-memory table to store the model.

11. Using the previously saved models, score each model on the validation data. The support vector machine is saved as an analytical store and the other models are saved as data tables. Therefore, the svm model requires the aStore action set to score the model, and the other models have their own scoring actions.

```
conn.loadActionSet('aStore')
actions = conn.builtins.help(actionSet='aStore')

#Score the support vector machine model
conn.aStore.score(
    table  = dict(name = indata, where = '_PartInd_ = 0'),
    rstore = "svm_model",
    out    = dict(name="svm_scored", replace=True)
)
```

Selected arguments:

Argument	Description
rstore	specifies the input analytical store.
out	specifies the in-memory table to store the scored data.

```
#Score the logistic regression model
lr_score_obj = conn.regression.logisticScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    restore = "lr_model",
    casout = dict(name="lr_scored", replace=True),
    copyVars = target
)

#Score the decision tree model
```

```
dt_score_obj = conn.decisionTree.dtreeScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    model = "dt_model",
    casout = dict(name="dt_scored",replace=True),
    copyVars = target,
    encodename = True,
    assessonerow = True
)

#Score the random forest model
rf_score_obj = conn.decisionTree.forestScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    model = "rf_model",
    casout = dict(name="rf_scored",replace=True),
    copyVars = target,
    encodename = True,
    assessonerow = True
)

#Score the gradient boosting model
gb_score_obj = conn.decisionTree.gbtreeScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    model = "gbt_model",
    casout = dict(name="gbt_scored",replace=True),
    copyVars = target,
    encodename = True,
    assessonerow = True
)

#Score the neural network model
nn_score_obj = conn.neuralNet.annScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    model = "nn_model",
    casout = dict(name="nn_scored",replace=True),
    copyVars = target,
    encodename = True,
    assessonerow = True
)
```

Selected arguments:

Argument	Description
model	specifies the table that contains the model information.
copyVars	specifies the variables to transfer from the input table to the output table.
encodename	specifies whether encoding the variable names in the generated output table adds the target variable name to the predicted probability variable name, such as the predicted probabilities of each response variable level.
assessonerow	When the value is set to true, predicted probabilities are added to the results table for the event levels. All event probabilities are included as separate columns and are used with the assess action.

```
display(nn_score_obj['OutputCasTables'])
nn_score_obj['ScoreInfo']
```

12. From the percentile action set, use the assess action to evaluate the scoring results for each model.

```
conn.loadActionSet('percentile')
actions = conn.builtins.help(actionSet='percentile')

# Change the name of the prediction variable for logistic regression
conn.dataStep.runCode(code='''
    data lr_scored;
        set lr_scored;
        rename _PRED_ = P_BAD1;
    run;
'''
)

# Add the target variable to the svm scored table
conn.dataStep.runCode(code='''
    data svm_scored;
        merge svm_scored(keep=P_BAD1) lr_scored(keep=BAD);
    run;
'''
)
```

The first DATA step simply changes the variable name for the predicted probabilities from **_PRED_** to **P_BAD1** to be consistent with the naming convention of the other models. The default variable name is the prefix **P_** followed by the target and level, **P_BAD1**.

```
# Create prediction variable name
assess_input = 'P_' + target + '1'

# Assess the logistic regression model
lr_assess_obj = conn.percentile.assess(
   table = 'lr_scored',
   inputs = assess_input,
   casout = dict(name="lr_assess",replace=True),
   response = target,
   event = "1"
)

# Assess the support vector machine model
svm_assess_obj = conn.percentile.assess(
   table = 'svm_scored',
   inputs = assess_input,
   casout = dict(name="svm_assess",replace=True),
   response = target,
   event = "1"
)

# Assess the decision tree model
dt_assess_obj = conn.percentile.assess(
   table = "dt_scored",
   inputs = assess_input,
   casout = dict(name="dt_assess",replace=True),
   response = target,
   event = "1"
)

# Assess the random forest model
rf_assess_obj = conn.percentile.assess(
   table = "rf_scored",
   inputs = assess_input,
   casout = dict(name="rf_assess",replace=True),
   response = target,
   event = "1"
)

#Assess the gradient boosting model
gb_assess_obj = conn.percentile.assess(
   table = "gbt_scored",
   inputs = assess_input,
   casout = dict(name="gbt_assess",replace=True),
   response = target,
   event = "1"
)

# Assess the neural network model
nn_assess_obj = conn.percentile.assess(
   table = "nn_scored",
   inputs = assess_input,
   casout = dict(name="nn_assess",replace=True),
   response = target,
   event = "1"
)
```
Selected arguments:

Argument	Description
event	specifies the formatted value of the response variable that represents the event.
inputs	specifies the input variable for the analysis. This is the predicted probability of the modeling level for binary classification.
response	specifies the original response variable for the model.
casOut	specifies the in-memory table to save the results.

```
nn_assess_obj['OutputCasTables']
```
Notice that the assess action saves two separate tables. The first is the name that is specified by the casOut argument. The second is the same name but it ends in _ROC. Each table saves different information to assess the model.
```
display(conn.table.fetch(table='nn_assess', to=5))

conn.table.fetch(table='nn_assess_ROC', to=5)
```
13. Use the defCasTable function to create an object reference to the ROC tables for each model. Then use the to.casDataFrame function to download the tables to analyze the results on the client.
```
lr_assess_ROC = conn.CASTable(name = "lr_assess_ROC")
lr_assess_ROC = lr_assess_ROC.to_frame()
lr_assess_ROC['Model'] = 'Logistic Regression'

svm_assess_ROC = conn.CASTable(name = "svm_assess_ROC")
svm_assess_ROC = svm_assess_ROC.to_frame()
svm_assess_ROC['Model'] = 'Support Vector Machine'

dt_assess_ROC = conn.CASTable(name = "dt_assess_ROC")
dt_assess_ROC = dt_assess_ROC.to_frame()
dt_assess_ROC['Model']= 'Decision Tree'

rf_assess_ROC = conn.CASTable(name = "rf_assess_ROC")
rf_assess_ROC = rf_assess_ROC.to_frame()
rf_assess_ROC['Model'] = 'Random Forest'

gbt_assess_ROC = conn.CASTable(name = "gbt_assess_ROC")
gbt_assess_ROC = gbt_assess_ROC.to_frame()
gbt_assess_ROC['Model'] = 'Gradient Boosting'

nn_assess_ROC = conn.CASTable(name = "nn_assess_ROC")
nn_assess_ROC = nn_assess_ROC.to_frame()
nn_assess_ROC['Model'] = 'Neural Network'
```
A new variable, **Model**, was added to each local data frame to specify the model name for the results.

14. Stack the ROC data frames into one data frame and then print the confusion matrix results at a 0.50 cutoff rate for the probability of an event.
```
df_assess = pd.DataFrame()
df_assess = pd.concat([lr_assess_ROC,svm_assess_ROC,dt_assess_ROC,
                         rf_assess_ROC,gbt_assess_ROC,nn_assess_ROC])
cutoff_index = df_assess['_Cutoff_']==0.5
compare = df_assess[cutoff_index].reset_index(drop=True)
compare[['Model','_TP_','_FP_','_FN_','_TN_']]
```

15. Use the **_ACC_** variable to print and compute the misclassification for each model.

```
compare['Misclassification'] = 1-compare['_ACC_']
miss = compare[compare['_Cutoff_']==0.5]
       [['Model','Misclassification']]
miss.sort_values('Misclassification')
```

16. Using ggplot, compare ROC curves and add the area under the curve to the plot legend.

```
plt.figure(figsize=(8,8))
plt.plot()
models = list(df_assess.Model.unique())
display(models)

# Iteratively add each curve to the plot
for X in models:
    tmp = df_assess[df_assess['Model']==X]
    plt.plot(tmp['_FPR_'],tmp['_Sensitivity_'], label=X+'
       (C=%0.2f)'%tmp['_C_'].mean())

plt.xlabel('False Positive Rate', fontsize=15)
plt.ylabel('True Positive Rate', fontsize=15)
plt.legend(loc='lower right', fontsize=15)
plt.show()
```

17. Create table references for the other assess results and download them to the client.

```
lr_assess_lift = defCasTable(conn, tablename = "lr_assess")
lr_assess_lift = to.casDataFrame(lr_assess_lift)
lr_assess_lift$Model = 'Logistic Regression'

svm_assess_lift = defCasTable(conn, tablename = "svm_assess")
svm_assess_lift = to.casDataFrame(svm_assess_lift)
svm_assess_lift$Model = 'Support Vector Machine'

dt_assess_lift = defCasTable(conn, tablename = "dt_assess")
dt_assess_lift = to.casDataFrame(dt_assess_lift)
dt_assess_lift$Model = 'Decision Tree'

rf_assess_lift = defCasTable(conn, tablename = "rf_assess")
rf_assess_lift = to.casDataFrame(rf_assess_lift)
rf_assess_lift$Model = 'Random Forest'

gbt_assess_lift = defCasTable(conn, tablename = "gbt_assess")
gbt_assess_lift = to.casDataFrame(gbt_assess_lift)
gbt_assess_lift$Model = 'Gradient Boosting'

nn_assess_lift = defCasTable(conn, tablename = "nn_assess")
nn_assess_lift = to.casDataFrame(nn_assess_lift)
nn_assess_lift$Model = 'Neural Network'

df_assess = rbind(lr_assess_lift, svm_assess_lift, dt_assess_lift, rf_assess_lift, gbt_assess_lift, nn_assess_lift)
```

18. Using matplotlib, create a lift curve.

```
plt.figure(figsize=(8,8))
plt.plot()
models = list(df_assess.Model.unique())
display(models)

# Iteratively add curves to the plot
for X in models:
    tmp = df_assess[df_assess['Model']==X]
    plt.plot(tmp['_Depth_'],tmp['_CumLift_'], label=X)

plt.xlabel('False Positive Rate', fontsize=15)
plt.ylabel('True Positive Rate', fontsize=15)
plt.legend(loc='upper right', fontsize=15)
plt.show()
```

19. Sample 75% of the **HMEQ** data set and then bring the data to the client. For larger data sets, you need to reduce the amount of data by creating a sample subset before you bring it to the client so that you do not use too much RAM. Then fit a gradient boosting model locally.

```
conn.sampling.srs(
    table   = indata,
    samppct = 75,
    seed    = 12345,
    partind = False,
    output  = dict(casOut = dict(name = 'mysam', replace = True),
                   copyVars = 'ALL')
)

# Bring data locally
mysam = conn.CASTable(name = "mysam")
df = mysam.to_frame()
df = df[[target]+inputs+['_PartInd_']]

# Create dummy variables
df = pd.concat([df, pd.get_dummies(df[nominals[1:]])],
                axis=1).drop(nominals[1:], axis=1)

# Split into training and validation
train = df[df['_PartInd_']==1]
```

```
valid = df[df['_PartInd_']==0]

# Split target and inputs
x_train = train.drop(target, axis=1)
x_valid = valid.drop(target, axis=1)
y_train = train[target]
y_valid = valid[target]

# Build python gradient boosting model with scikit-learn
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

# Fit the model locally
gb = GradientBoostingClassifier()
gb.fit(x_train, y_train)
```

20. Print the misclassification rate for the local model and then combine and print misclassification rates for all models, both local and CAS.

```
gb_score = gb.predict(x_valid)
gb_misclassification = 1 - accuracy_score(y_valid, gb_score)
pymiss = pd.DataFrame({'Model':'Python - Gradient Boosting',
        'Misclassification':gb_misclassification}, index=[0])
pymiss

err = data.frame(rbind(miss, xgb))
err[,-1] = round(as.numeric(as.character(err[,-1])),7)
err = err[order(err[,-1]),]
rownames(err) = NULL
err
```

Because the CAS actions results are local, they can be combined with the local modeling results to create a data frame for comparison.

21. Use the tableInfo action from the table action set to view all the tables that are currently on the server.

```
conn.table.tableInfo()['TableInfo'][['Name','Rows','Columns']]
```

Note: Unless they are saved or promoted, all tables are deleted after the CAS session is terminated.

22. Use the caslibInfo action to view the path for the default caslib **CASUSER**. Then use the addCaslib action to create a user-defined caslb named **mycl** located in the home directory.

```
conn.table.caslibInfo(active=False, caslib="casuser")

cconn.table.addCaslib(name="mycl", path=os.environ.get("HOME"),
     dataSource="PATH", description="Personal File Save Location",
     activeOnAdd=False)
```

Selected arguments:

Argument	Description
name	specifies the name of the caslib to be added.
dataSource	specifies the data source type.
path	specifies the data source specific information.
description	specifies a string description of the new caslib.
activeOnAdd	When the value is set to *true*, the new caslib becomes the active caslib. When it is set to *false*, the default caslib remains active.

The new caslib is local, which means that the connection to the directory is terminated when the session ends. However, the data that is saved in the library are permanent. The new caslib is not active, which means that the name must be specified so that it can be used. Otherwise, the default caslib is assumed.

```
conn.table.caslibInfo()
```

23. Use the save action to save the gradient boosting model on the server in the **mycl** caslib. Also, using the attribute action, save the table attributes for the model.

```
conn.table.save(caslib = 'mycl', table = dict(name = 'gbt_model'),
                name = 'best_model_gbt', replace = True)

conn.table.attribute(caslib = 'CASUSER',
    table = 'gbt_model_attr', name = 'gbt_model', task='convert')

conn.table.save(caslib = 'mycl', table = 'gbt_model_attr',
                name = 'attr', replace = True)
```

Selected argument:

Argument	Description
task	specifies the task to perform. The convert task creates a table for the attributes so that they can be saved.

24. Using the dropCaslib action, drop the user-defined caslb, mycl.

```
conn.table.dropCaslib(caslib="mycl")
```

25. Promote the **HMEQ** data set to the global scope. This causes the table to persist on the server unless it is explicitly dropped. It can be accessed from other APIs and other sessions.

```
conn.table.promote(caslib="casuser", name=indata)
conn.table.tableInfo()
```

26. End the session and disconnect from CAS.

```
conn.session.endSession()
```