Performance and Prediction

Model Performance

Given a trained H2O model, the `h2o.performance()` (R)/`model_performance()` (Python) function computes a model's performance on a given dataset.

Notes:

- If the provided dataset does not contain the response/target column from the model object, no performance will be returned. Instead, a warning message will be printed.
- For binary classification problems, H2O uses the model along with the given dataset to calculate the threshold that will give the maximum F1 for the given dataset.

This section describes how H2O-3 can be used to evaluate model performance. Models can also be evaluated with specific model metrics, stopping metrics, and performance graphs.

Evaluation Model Metrics

H2O-3 provides a variety of metrics that can be used for evaluating supervised and unsupervised models. The metrics for this section only cover supervised learning models, which vary based on the model type (classification or regression).

Regression

The following evaluation metrics are available for regression models. (Note that H2O-3 also calculates regression metrics for Classification problems.)

- R2 (R Squared)
- MSE (Mean Squared Error)
- RMSE (Root Mean Squared Error)
- RMSLE (Root Mean Squared Logarithmic Error)
- MAE (Mean Absolute Error)

Each metric is described in greater detail in the sections that follow. The examples are based off of a GBM model built using the **cars_20mpg.csv** dataset.

R | **Python**

```
import h2o
from h2o.estimators.gbm import H2OGradientBoostingEstimator
h2o.init()

# import the cars dataset:
# this dataset is used to classify whether or not a car is economical based on
# the car's displacement, power, weight, and acceleration, and the year it was made
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the predictor names and the response column name
predictors = ["displacement","power","weight","acceleration","year"]
response = "cylinders"

# split into train and validation sets
train, valid = cars.split_frame(ratios = [.8], seed = 1234)

# train a GBM model
cars_gbm = H2OGradientBoostingEstimator(distribution = "poisson", seed = 1234)
cars_gbm.train(x = predictors,
               y = response,
               training_frame = train,
               validation_frame = valid)

# retrieve the model performance
perf = cars_gbm.model_performance(valid)
perf
```

R2 (R Squared)

The R2 value represents the degree that the predicted value and the actual value move in unison. The R2 value varies between 0 and 1 where 0 represents no correlation between the predicted and actual value and 1 represents complete correlation.

Example

Using the previous example, run the following to retrieve the R2 value.

R | Python

```
# retrieve the r2 value:
cars_gbm.r2()
0.9930650688408735

# retrieve the r2 value for the validation data:
cars_gbm.r2(valid=True)
0.9886704207301097
```

MSE (Mean Squared Error)

The MSE metric measures the average of the squares of the errors or deviations. MSE takes the distances from the points to the regression line (these distances are the "errors") and squaring them to remove any negative signs. MSE incorporates both the variance and the bias of the predictor.

MSE also gives more weight to larger differences. The bigger the error, the more it is penalized. For example, if your correct answers are 2,3,4 and the algorithm guesses 1,4,3, then the absolute error on each one is exactly 1, so squared error is also 1, and the MSE is 1. But if the algorithm guesses 2,3,6, then the errors are 0,0,2, the squared errors are 0,0,4, and the MSE is a higher 1.333. The smaller the MSE, the better the model's performance. (**Tip**: MSE is sensitive to outliers. If you want a more robust metric, try mean absolute error (MAE).)

MSE equation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Example

Using the previous example, run the following to retrieve the MSE value.

R | Python

```
# retrieve the mse value:
cars_gbm.mse()
0.019173269728097173

# retrieve the mse value for the validation data:
cars_gbm.mse(valid=True)
0.03769791966551617
```

RMSE (Root Mean Squared Error)

The RMSE metric evaluates how well a model can predict a continuous value. The RMSE units are the same as the predicted target, which is useful for understanding if the size of the error is of concern or not. The smaller the RMSE, the better the model's performance. (**Tip**: RMSE is sensitive to outliers. If you want a more robust metric, try mean absolute error (MAE).)

RMSE equation:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

Where:

- N is the total number of rows (observations) of your corresponding dataframe.
- y is the actual target value.
- $\hat{y}$ is the predicted target value.

Example

Using the previous example, run the following to retrieve the RMSE value.

R | Python

```
# retrieve the rmse value:
cars_gbm.rmse()
0.13846757645057983

# retrieve the rmse value for the validation data:
cars_gbm.rmse(valid=True)
0.19415952118172358
```

RMSLE (Root Mean Squared Logarithmic Error)

This metric measures the ratio between actual values and predicted values and takes the log of the predictions and actual values. Use this instead of RMSE if an under-prediction is worse than an over-prediction. You can also use this when you don't want to penalize large differences when both of the values are large numbers.

RMSLE equation:

$$RMSLE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(ln\left(\frac{y_i + 1}{\hat{y}_i + 1}\right)\right)^2}$$

Where:

- N is the total number of rows (observations) of your corresponding dataframe.
- y is the actual target value.
- $\hat{y}$ is the predicted target value.

Example

Using the previous example, run the following to retrieve the RMSLE value.

R | Python

```
# retrieve the rmsle value:
cars_gbm.rmsle()
0.023320830800314333

# retrieve the rmsle value for the validation data:
cars_gbm.rmsle(valid=True)
0.03359130162278705
```

MAE (Mean Absolute Error)

The mean absolute error is an average of the absolute errors. The MAE units are the same as the predicted target, which is useful for understanding whether the size of the error is of concern or not. The smaller the MAE the better the model's performance. (**Tip**: MAE is robust to outliers. If you want a metric that is sensitive to outliers, try root mean squared error (RMSE).)

MAE equation:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|x_i - x|$$

Where:

- N is the total number of errors
- $|x_i - x|$ equals the absolute errors.

Example

Using the previous example, run the following to retrieve the MAE value.

R | Python

```
# retrieve the mae value:
cars_gbm.mae()
0.06140515094616347

# retrieve the mae value for the validation data:
cars_gbm.mae(valid=True)
0.07947861719967757
```

Classification

H2O-3 calculates regression metrics for classification problems. The following additional evaluation metrics are available for classification models:

- Gini Coefficient
- Absolute MCC (Matthews Correlation Coefficient)
- F1
- F0.5
- F2
- Accuracy
- Logloss
- AUC (Area Under the ROC Curve)
- AUCPR (Area Under the Precision-Recall Curve)
- Kolmogorov-Smirnov (KS) Metric

Each metric is described in greater detail in the sections that follow. The examples are based off of a GBM model built using the **allyears2k_headers.zip** dataset.

R | **Python**

```
import h2o
from h2o.estimators.gbm import H2OGradientBoostingEstimator
h2o.init()

# import the airlines dataset:
# This dataset is used to classify whether a flight will be delayed 'YES' or not "NO"
# original data can be found at http://www.transtats.bts.gov/
airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/airlines/allyears2k_headers.zip")

# convert columns to factors
airlines["Year"]= airlines["Year"].asfactor()
airlines["Month"]= airlines["Month"].asfactor()
airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
airlines["Cancelled"] = airlines["Cancelled"].asfactor()
airlines['FlightNum'] = airlines['FlightNum'].asfactor()

# set the predictor names and the response column name
predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
              "DayOfWeek", "Month", "Distance", "FlightNum"]
response = "IsDepDelayed"

# split into train and validation sets
train, valid = airlines.split_frame(ratios = [.8], seed = 1234)

# train your model
airlines_gbm = H2OGradientBoostingEstimator(sample_rate = .7, seed = 1234)
airlines_gbm.train(x = predictors,
                   y = response,
                   training_frame = train,
                   validation_frame = valid)

# retrieve the model performance
perf = airlines_gbm.model_performance(valid)
perf
```

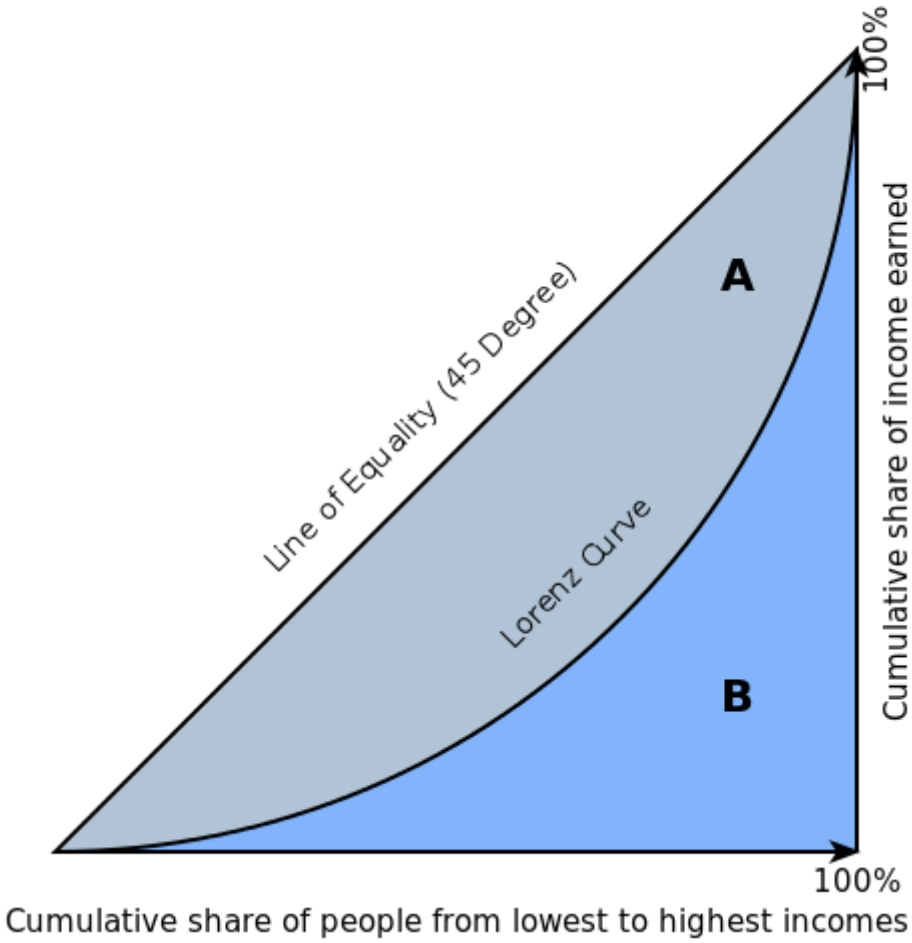
Gini Coefficient

The Gini index is a well-established method to quantify the inequality among values of a frequency distribution, and can be used to measure the quality of a binary classifier. A Gini index of zero expresses perfect equality (or a totally useless classifier), while a Gini index of one expresses maximal inequality (or a perfect classifier).

The Gini index is based on the Lorenz curve. The Lorenz curve plots the true positive rate (y-axis) as a function of percentiles of the population (x-axis).

The Lorenz curve represents a collective of models represented by the classifier. The location on the curve is given by the probability threshold of a particular model. (i.e., Lower probability thresholds for classification typically lead to more true positives, but also to more false positives.)

The Gini index itself is independent of the model and only depends on the Lorenz curve determined by the distribution of the scores (or probabilities) obtained from the classifier.



Example

Using the previous example, run the following to retrieve the Gini coefficient value.

R | Python

```
# retrieve the gini coefficient:
perf.gini()
0.48299402265152613

# retrieve the gini coefficient for both the training and validation data:
airlines_gbm.gini(train=True, valid=True, xval=False)
{u'train': 0.5715841348613386, u'valid': 0.48299402265152613}
```

Absolute MCC (Matthews Correlation Coefficient)

Setting the `absolute_mcc` parameter sets the threshold for the model's confusion matrix to a value that generates the highest Matthews Correlation Coefficient. The MCC score provides a measure of how well a binary classifier detects true and false positives, and true and false negatives. The MCC is called a correlation coefficient because it indicates how correlated the actual and predicted values are; 1 indicates a perfect classifier, -1 indicates a classifier that predicts the opposite class from the actual value, and 0 means the classifier does no better than random guessing.

$$MCC = \frac{TP\ x\ TN\ - FP\ x\ FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

Example

Using the previous example, run the following to retrieve the MCC value.

R | Python

```
# retrieve the mcc for the performance object:
perf.mcc()
[0.5426977730968023, 0.36574105494931725]]

# retrieve the mcc for both the training and validation data:
airlines_gbm.mcc(train=True, valid=True, xval=False)
{u'train': [[0.5203060957871319, 0.42414048381779923]], u'valid': [[0.5426977730968023,
0.36574105494931725]]}
```

F1

The F1 score provides a measure for how well a binary classifier can classify positive cases (given a threshold value). The F1 score is calculated from the harmonic mean of the precision and recall. An F1 score of 1 means both precision and recall are perfect and the model correctly identified all the positive cases and didn't mark a negative case as a positive case. If either precision or recall are very low it will be reflected with a F1 score closer to 0.

$$F1 = 2\left(\frac{(precision)\ (recall)}{precision + recall}\right)$$

Where:

- *precision* is the positive observations (true positives) the model correctly identified from all the observations it labeled as positive (the true positives + the false positives).
- *recall* is the positive observations (true positives) the model correctly identified from all the actual positive cases (the true positives + the false negatives).

Example

Using the previous example, run the following to retrieve the F1 value.

R | Python

```
# retrieve the F1 coefficient for the performance object:
perf.F1()
[[0.35417599264806404, 0.7228980805623143]]

# retrieve the F1 coefficient for both the training and validation data:
airlines_gbm.F1(train=True, valid=True, xval=False)
{u'train': [[0.3869697386893616, 0.7451099672437997]], u'valid': [[0.35417599264806404,
0.7228980805623143]]}
```

F0.5

The F0.5 score is the weighted harmonic mean of the precision and recall (given a threshold value). Unlike the F1 score, which gives equal weight to precision and recall, the F0.5 score gives more weight to precision than to recall. More weight should be given to precision for cases where False Positives are considered worse than False Negatives. For example, if your use case is to predict which products you will run out of, you may consider False Positives worse than False Negatives. In this case, you want your predictions to be very precise and only capture the products that will definitely run out. If you predict a product will need to be restocked when it actually doesn't, you incur cost by having purchased more inventory than you actually need.

F0.5 equation:

$$F0.5 = 1.25 \left(\frac{(precision)\ (recall)}{0.25\ precision + recall} \right)$$

Where:

- *precision* is the positive observations (true positives) the model correctly identified from all the observations it labeled as positive (the true positives + the false positives).
- *recall* is the positive observations (true positives) the model correctly identified from all the actual positive cases (the true positives + the false negatives).

Example

Using the previous example, run the following to retrieve the F0.5 value.

R | Python

```
# retrieve the F1 coefficient for the performance object:
perf.F0point5()
[[0.5426977730968023, 0.7047449127206096]]

# retrieve the F1 coefficient for both the training and validation data:
airlines_gbm.F0point5(train=True, valid=True, xval=False)
{u'train': [[0.5529885092975969, 0.7331482319556736]], u'valid': [[0.5426977730968023,
0.7047449127206096]]}
```

F2

The F2 score is the weighted harmonic mean of the precision and recall (given a threshold value). Unlike the F1 score, which gives equal weight to precision and recall, the F2 score gives more weight to recall (penalizing the model more for false negatives then false positives). An F2 score ranges from 0 to 1, with 1 being a perfect model.

$$F2 = 5\left(\frac{(precision)\,(recall)}{4\,precision + recall}\right)$$

Example

Using the previous example, run the following to retrieve the F2 value.

R | Python

```
# retrieve the F2 coefficient for the performance object:
perf.F2()
[[0.1957813426628461, 0.8502311018339048]]

# retrieve the F2 coefficient for both the training and validation data:
airlines_gbm.F2(train=True, valid=True, xval=False)
{u'train': [[0.24968434313831914, 0.8548787509793371]], u'valid': [[0.1957813426628461,
0.8502311018339048]]}
```

Accuracy

In binary classification, Accuracy is the number of correct predictions made as a ratio of all predictions made. In multiclass classification, the set of labels predicted for a sample must exactly match the corresponding set of labels in y_true.

Accuracy equation:

$$Accuracy = \left(\frac{\text{number correctly predicted}}{\text{number of observations}} \right)$$

Example

Using the previous example, run the following to retrieve the Accurace value.

R | Python

```
# retrieve the accuracy coefficient for the performance object:
perf.accuracy()
[[0.5231232172827827, 0.6816775524235132]]

# retrieve the accuracy coefficient for both the training and validation data:
airlines_gbm.accuracy(train=True, valid=True, xval=False)
{u'train': [[0.5164521833040745, 0.7118095940540694]], u'valid': [[0.5231232172827827,
0.6816775524235132]]}
```

Logloss

The logarithmic loss metric can be used to evaluate the performance of a binomial or multinomial classifier. Unlike AUC which looks at how well a model can classify a binary target, logloss evaluates how close a model's predicted values (uncalibrated probability estimates) are to the actual target value. For example, does a model tend to assign a high predicted value like .80 for the positive class, or does it show a poor ability to recognize the positive class and assign a lower predicted value like .50? Logloss can be any value greater than or equal to 0, with 0 meaning that the model correctly assigns a probability of 0% or 100%.

Binary classification equation:

$$Logloss = -\frac{1}{N}\sum_{i=1}^{N} w_i(\, y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i) \,)$$

Multiclass classification equation:

$$Logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} w_i(\, y_{i,j} \, \ln(p_{i,j}) \,)$$

Where:

- *N* is the total number of rows (observations) of your corresponding dataframe.
- *w* is the per row user-defined weight (defaults is 1).
- *C* is the total number of classes (C=2 for binary classification).
- *p* is the predicted value (uncalibrated probability) assigned to a given row (observation).
- *y* is the actual target value.

Example

Using the previous example, run the following to retrieve the logloss value.

R | Python

```
# retrieve the logloss for the performance object:
perf.logloss()
0.5967028742962095

# retrieve the logloss for both the training and validation data:
airlines_gbm.logloss(train=True, valid=True, xval=False)
{u'train': 0.5607154587919981, u'valid': 0.5967028742962095}
```

AUC (Area Under the ROC Curve)

This model metric is used to evaluate how well a binary classification model is able to distinguish between true positives and false positives. An AUC of 1 indicates a perfect classifier, while an AUC of .5 indicates a poor classifier, whose performance is no better than random guessing.

H2O uses the trapezoidal rule to approximate the area under the ROC curve. (**Tip**: AUC is usually not the best metric for an imbalanced binary target because a high number of True Negatives can cause the AUC to look inflated. For an imbalanced binary target, we recommend AUCPR or MCC.)

Example

Using the previous example, run the following to retrieve the AUC.

R | Python

```
# retrieve the AUC for the performance object:
perf.auc()
0.7414970113257631

# retrieve the AUC for both the training and validation data:
airlines_gbm.auc(train=True, valid=True, xval=False)
{u'train': 0.7857920674306693, u'valid': 0.7414970113257631}
```

AUCPR (Area Under the Precision-Recall Curve)

This model metric is used to evaluate how well a binary classification model is able to distinguish between precision recall pairs or points. These values are obtained using different thresholds on a probabilistic or other continuous-output classifier. AUCPR is an average of the precision-recall weighted by the probability of a given threshold.

The main difference between AUC and AUCPR is that AUC calculates the area under the ROC curve and AUCPR calculates the area under the Precision Recall curve. The Precision Recall curve does not care about True Negatives. For imbalanced data, a large quantity of True Negatives usually overshadows the effects of changes in other metrics like False Positives. The AUCPR will be much more sensitive to True Positives, False Positives, and False Negatives than AUC. As such, AUCPR is recommended over AUC for highly imbalanced data.

Example

Using the previous example, run the following to retrieve the AUCPR.

R | Python

```
# retrieve the AUCPR for the performance object:
perf.aucpr()
0.7609887253334723

# retrieve the AUCPR for both the training and validation data:
airlines_gbm.aucpr(train=True, valid=True, xval=False)
{u'train': 0.801959918132391, u'valid': 0.7609887253334723}
```

Multinomial AUC (Area Under the ROC Curve)

This model metric is used to evaluate how well a multinomial classification model is able to distinguish between true positives and false positives across all domains. The metric is composed of these outputs:

One class versus one class (OVO) AUCs - calculated for all pairwise combination of classes ((number of classes × number of classes / 2) - number of classes results)

One class versus rest classes (OVR) AUCs - calculated for all combination one class and rest of classes (number of classes results)

Macro average OVR AUC - Uniformly weighted average of all OVO AUCs

$$\frac{1}{c}\sum_{j=1}^{c}\mathrm{AUC}(j, rest_j)$$

where c is the number of classes and $\mathrm{AUC}(j, rest_j)$ is the AUC with class j as the positive class and rest classes $rest_j$ as the negative class. The result AUC is normalized by number of classes.

Weighted average OVR AUC - Prevalence weighted average of all OVR AUCs

$$\frac{1}{\sum_{j=1}^{c} p(j)}\sum_{j=1}^{c} p(j)\mathrm{AUC}(j, rest_j)$$

where c is the number of classes, $\mathrm{AUC}(j, rest_j)$ is the AUC with class j as the positive class and rest classes $rest_j$ as the negative class and $p(j)$ is the prevalence of class j (number of positives of class j). The result AUC is normalized by sum of all weights.

Macro average OVO AUC - Uniformly weighted average of all OVO AUCs

$$\frac{2}{c}\sum_{j=1}^{c}\sum_{k\neq j}^{c}\frac{1}{2}(\mathrm{AUC}(j|k) + \mathrm{AUC}(k|j))$$

where c is the number of classes and $\mathrm{AUC}(j, k)$ is the AUC with class j as the positive class and class k as the negative class. The result AUC is normalized by number of all class combinations.

Weighted average OVO AUC - Prevalence weighted average of all OVO AUCs

$$\frac{2}{\sum_{j=1}^{c}\sum_{k\neq j}^{c} p(j \cup k)}\sum_{j=1}^{c}\sum_{k\neq j}^{c} p(j \cup k)\frac{1}{2}(\mathrm{AUC}(j|k) + \mathrm{AUC}(k|j))$$

where c is the number of classes, $\mathrm{AUC}(j, k)$ is the AUC with class j as the positive class and class k as the negative class and $p(j \cup k)$ is prevalence of class j and class k (sum of positives of both classes). The result AUC is normalized by sum of all weights.

Result Multinomial AUC table could look for three classes like this:

Note Macro and weighted average values could be the same if the classes are same distributed.

type	first_class_domain	second_class_domain	auc
1 vs Rest	1	None	0.996891
2 vs Rest	2	None	0.996844
3 vs Rest	3	None	0.987593
Macro OVR	None	None	0.993776
Weighted OVR	None	None	0.993776
1 vs 2	1	2	0.969807
1 vs 3	1	3	1.000000
2 vs 3	2	3	0.995536
Macro OVO	None	None	0.988447
Weighted OVO	None	None	0.988447

Default value of AUC

Multinomial AUC metric can be used for early stopping and during grid search as binomial AUC. In case of Multinomial AUC only one value need to be specified. The AUC calculation is disabled (set to `NONE`) by default. However this option can be changed using `auc_type` model parameter to any other average type of AUC and AUCPR - `MACRO_OVR`, `WEIGHTED_OVR`, `MACRO_OVO`, `WEIGHTED_OVO`.

Example

R | **Python**

```
import h2o
from h2o.estimators.gbm import H2OGradientBoostingEstimator
h2o.init()

# import the cars dataset:
# this dataset is used to classify whether or not a car is economical based on
# the car's displacement, power, weight, and acceleration, and the year it was made
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the predictor names and the response column name
predictors = ["displacement","power","weight","acceleration","year"]
response = "cylinders"
cars[response] = cars[response].asfactor()

# split into train and validation sets
train, valid = cars.split_frame(ratios = [.8], seed = 1234)

# train a GBM model
cars_gbm = H2OGradientBoostingEstimator(distribution = "multinomial", seed = 1234)
cars_gbm.train(x = predictors,
               y = response,
               training_frame = train,
               validation_frame = valid)

# get result on training data from h2o
h2o_auc_table = cars_gbm.multinomial_auc_table(train)
print(h2o_auc_table)

# get default value
h2o_default_auc = cars_gbm.auc()
print(h2o_default_auc)
```

Notes - Calculation of this metric can be very expensive on time and memory when the domain is big. So it is disabled by default. - To enable it setup system property `sys.ai.h2o.auc.maxClasses` to a number.

Multinomial AUCPR (Area Under the Precision-Recall Curve)

This model metric is used to evaluate how well a multinomial classification model is able to distinguish between precision recall pairs or points across all domains. The metric is composed of these outputs:

One class versus one class (OVO) AUCPRs - calculated for all pairwise AUCPR combination of classes ((number of classes × number of classes / 2) - number of classes results)

One class versus rest classes (OVR) AUCPRs - calculated for all combination one class and rest of classes AUCPR (number of classes results)

Macro average OVR AUCPR - Uniformly weighted average of all OVR AUCPRs

$$\frac{1}{c}\sum_{j=1}^{c}\mathrm{AUCPR}(j, rest_j)$$

where c is the number of classes and $\mathrm{AUCPR}(j, rest_j)$ is the AUCPR with class j as the positive class and rest classes $rest_j$ as the negative class. The result AUCPR is normalized by number of classes.

Weighted average OVR AUCPR - Prevalence weighted average of all OVR AUCPRs

$$\frac{1}{\sum_{j=1}^{c} p(j)}\sum_{j=1}^{c} p(j)\mathrm{AUCPR}(j, rest_j)$$

where c is the number of classes, $\mathrm{AUCPR}(j, rest_j)$ is the AUCPR with class j as the positive class and rest classes $rest_j$ as the negative class and $p(j)$ is the prevalence of class j (number of positives of class j). The result AUCPR is normalized by sum of all weights.

Macro average OVO AUCPR - Uniformly weighted average of all OVO AUCPRs

$$\frac{2}{c}\sum_{j=1}^{c}\sum_{k\neq j}^{c}\frac{1}{2}(\mathrm{AUCPR}(j|k) + \mathrm{AUCPR}(k|j))$$

where c is the number of classes and $\mathrm{AUCPR}(j, k)$ is the AUCPR with class j as the positive class and class k as the negative class. The result AUCPR is normalized by number of all class combinations.

Weighted average OVO AUCPR - Prevalence weighted average of all OVO AUCPRs

$$\frac{2}{\sum_{j=1}^{c}\sum_{k\neq j}^{c} p(j \cup k)}\sum_{j=1}^{c}\sum_{k\neq j}^{c} p(j \cup k)\frac{1}{2}(\mathrm{AUCPR}(j|k) + \mathrm{AUCPR}(k|j))$$

where c is the number of classes, $\mathrm{AUCPR}(j, k)$ is the AUCPR with class j as the positive class and class k as the negative class and $p(j \cup k)$ is prevalence of class j and class k (sum of positives of both classes). The result AUCPR is normalized by sum of all weights.

Result Multinomial AUCPR table could look for three classes like this:

Note Macro and weighted average values could be the same if the classes are same distributed.

type	first_class_domain	second_class_domain	aucpr
1 vs Rest	1	None	0.996891
2 vs Rest	2	None	0.996844
3 vs Rest	3	None	0.987593
Macro OVR	None	None	0.993776
Weighted OVR	None	None	0.993776
1 vs 2	1	2	0.969807
1 vs 3	1	3	1.000000
2 vs 3	2	3	0.995536
Macro OVO	None	None	0.988447
Weighted OVO	None	None	0.988447

Default value of AUCPR

Multinomial AUCPR metric can be also used for early stopping and during grid search as binomial AUCPR. In case of Multinomial AUCPR only one value need to be specified. The AUCPR calculation is disabled (set to `NONE`) by default. However this option can be changed using `auc_type` model parameter to any other average type of AUC and AUCPR - `MACRO_OVR`, `WEIGHTED_OVR`, `MACRO_OVO`, `WEIGHTED_OVO`.

Example

R | **Python**

```
import h2o
from h2o.estimators.gbm import H2OGradientBoostingEstimator
h2o.init()

# import the cars dataset:
# this dataset is used to classify whether or not a car is economical based on
# the car's displacement, power, weight, and acceleration, and the year it was made
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the predictor names and the response column name
predictors = ["displacement","power","weight","acceleration","year"]
response = "cylinders"
cars[response] = cars[response].asfactor()

# split into train and validation sets
train, valid = cars.split_frame(ratios = [.8], seed = 1234)

# train a GBM model
cars_gbm = H2OGradientBoostingEstimator(distribution = "multinomial", seed = 1234)
cars_gbm.train(x = predictors,
               y = response,
               training_frame = train,
               validation_frame = valid)

# get result on training data from h2o
h2o_aucpr_table = cars_gbm.multinomial_aucpr_table(train)
print(h2o_aucpr_table)

# get default value
h2o_default_aucpr = cars_gbm.aucpr()
print(h2o_default_aucpr)
```

Notes - Calculation of this metric can be very expensive on time and memory when the domain is big. So it is disabled by default. - To enable it setup system property `sys.ai.h2o.auc.maxClasses` to a number of maximum allowed classes.

Kolmogorov-Smirnov (KS) Metric

The Kolmogorov-Smirnov (KS) metric represents the degree of separation between the positive (1) and negative (0) cumulative distribution functions for a binomial model. It is a nonparametric test that compares the cumulative distributions of two unmatched data sets and does not assume that data are sampled from any defined distributions. The KS metric has more power to detect changes in the shape of the distribution and less to detect a shift in the median because it tests for more deviations from the null hypothesis. Detailed metrics per each group can be found in the gains-lift table.

Kolmogorov-Smirnov Equation:

$$KS = \sup_x | F_{1,n}(x) - F_{2,m}(x) |$$

Where:

- sup_x is the supremum function.
- $F_{1,n}$ is the sum of all events observed so far up to the bin *i* divided by the total number of events.
- $F_{2,m}$ is the sum of all non-events observed so far up to the bin *i* divided by the total number of non-events.

Examples

Using the previously imported and split airlines dataset, run the following to retrieve the KS metric.

R | Python

```
# build a new model using gainslift_bins:
model = H2OGradientBoostingEstimator(ntrees=1, gainslift_bins=10)
model.train(x=["Origin", "Distance"], y="IsDepDelayed", training_frame=train)

# retrieve the ks metric:
ks = model.kolmogorov_smirnov()
ks
0.20072346203696562
```

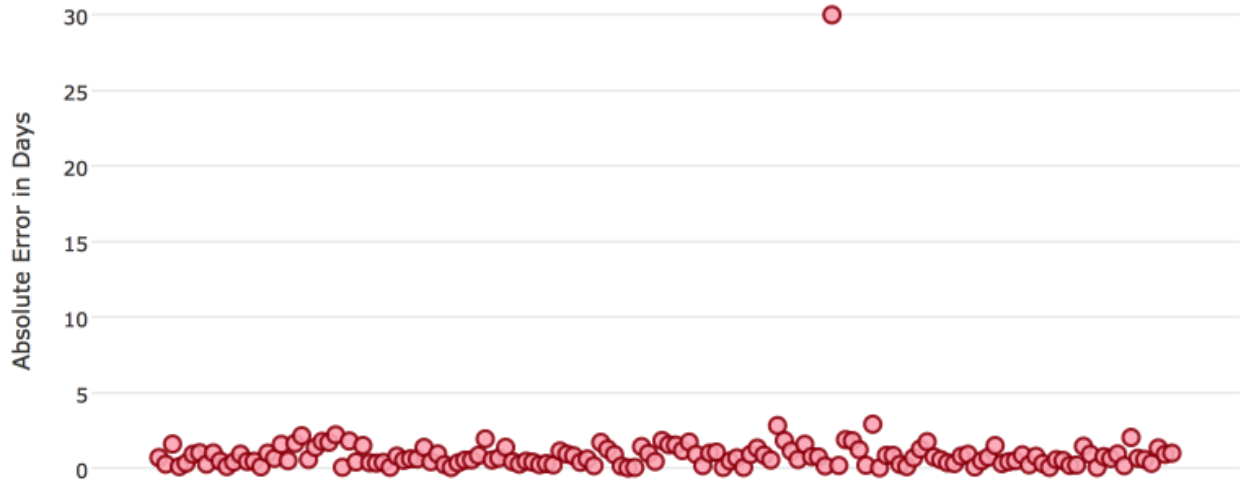
Metric Best Practices - Regression

When deciding which metric to use in a regression problem, some main questions to ask are:

- Do you want your metric sensitive to outliers?
- What unit should the metric be in?

Sensitive to Outliers

Certain metrics are more sensitive to outliers. When a metric is sensitive to outliers, it means that it is important that the model predictions are never “very” wrong. For example, let’s say we have an experiment predicting number of days until an event. The graph below shows the absolute error in our predictions.



Usually our model is very good. We have an absolute error less than 1 day about 70% of the time. There is one instance, however, where our model did very poorly. We have one prediction that was 30 days off.

Instances like this will more heavily penalize metrics that are sensitive to outliers. If you do not care about these outliers in poor performance as long as you typically have a very accurate prediction, then you would want to select a metric that is robust to outliers. You can see this reflected in the behavior of the metrics: `MSE` and `RMSE`.

	MSE	RMSE
Outlier	0.99	2.64
No Outlier	0.80	1.0

Calculating the `RMSE` and `MSE` on our error data, the `RMSE` is more than twice as large as the `MSE` because `RMSE` is sensitive to outliers. If you remove the one outlier record from our calculation, `RMSE` drops down significantly.

Performance Units

Different metrics will show the performance of your model in different units. Let's continue with our example where our target is to predict the number of days until an event. Some possible performance units are:

- Same as target: The unit of the metric is in days
    - ex: MAE = 5 means the model predictions are off by 5 days on average
- Percent of target: The unit of the metric is the percent of days

    - ex: MAPE = 10% means the model predictions are off by 10 percent on average
- Square of target: The unit of the metric is in days squared
    - ex: MSE = 25 means the model predictions are off by 5 days on average (square root of 25 = 5)

Comparison

Metric	Units	Sensitive to Outliers	Tip
R2	scaled between 0 and 1	No	use when you wan
MSE	square of target	Yes	
RMSE	same as target	Yes	
RMSLE	log of target	Yes	
RMSPE	percent of target	Yes	use when target va
MAE	same as target	No	
MAPE	percent of target	No	use when target va
SMAPE	percent of target divided by 2	No	use when target va

Metric Best Practices - Classification

When deciding which metric to use in a classification problem some main questions to ask are:

- Do you want the metric to evaluate the predicted probabilities or the classes that those probabilities can be converted to?
- Is your data imbalanced?

Does the Metric Evaluate Probabilities or Classes?

The final output of a model is a predicted probability that a record is in a particular class. The metric you choose will either evaluate how accurate the probability is or how accurate the assigned class is from that probability.

Choosing this depends on the use of the model. Do you want to use the probabilities, or do you want to convert those probabilities into classes? For example, if you are predicting whether a customer will churn, you can take the predicted probabilities and turn them into classes - customers who will churn vs customers who won't churn. If you are predicting the expected loss of revenue, you will instead use the predicted probabilities (predicted probability of churn * value of customer).

If your use case requires a class assigned to each record, you will want to select a metric that evaluates the model's performance based on how well it classifies the records. If your use case will use the probabilities, you will want to select a metric that evaluates the model's performance based on the predicted probability.

Is the Metric Robust to Imbalanced Data?

For certain use cases, positive classes may be very rare. In these instances, some metrics can be misleading. For example, if you have a use case where 99% of the records have `Class = No`, then a model that always predicts `No` will have 99% accuracy.

For these use cases, it is best to select a metric that does not include True Negatives or considers relative size of the True Negatives like AUCPR or MCC.

Metric Comparison

Metric	Evaluation Based On	Tip
MCC	Predicted class	good for imbalanced data
F1	Predicted class	
F0.5	Predicted class	good when you want to give more weight to precisi
F2	Predicted class	good when you want to give more weight to recall
Accuracy	Predicted class	highly interpretable, bad for imbalanced data
Logloss	Predicted value	
AUC	Predicted value	good for imbalanced data
AUCPR	Predicted value	good for imbalanced data

Stopping Model Metrics

Stopping metric parameters are specified in conjunction with a stopping tolerance and a number of stopping rounds. A metric specified with the stopping_metric option specifies the metric to consider when early stopping is specified.

Misclassification

This parameter specifies that a model must improve its misclassification rate by a given amount (specified by the stopping_tolerance parameter) in order to continue iterating. The misclassification rate is the number of observations incorrectly classified divided by the total number of observations.

Examples:

R | Python

```
# import H2OGradientBoostingEstimator and the airlines dataset:
from h2o.estimators import H2OGradientBoostingEstimator
airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/airlines/allyears2k_headers.zip")

# set the factors:
airlines["Year"]= airlines["Year"].asfactor()
airlines["Month"]= airlines["Month"].asfactor()
airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
airlines["Cancelled"] = airlines["Cancelled"].asfactor()
airlines['FlightNum'] = airlines['FlightNum'].asfactor()

# set the predictors and response columns:
predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
              "DayOfWeek", "Month", "Distance", "FlightNum"]
response = "IsDepDelayed"

# split the data into training and validation sets:
train, valid= airlines.split_frame(ratios = [.8], seed = 1234)

# build and train the model using the misclassification stopping metric:
airlines_gbm = H2OGradientBoostingEstimator(stopping_metric = "misclassification",
                                            stopping_rounds = 3,
                                            stopping_tolerance = 1e-2,
                                            seed = 1234)
airlines_gbm.train(x = predictors, y = response,
                   training_frame = train, validation_frame = valid)

# retrieve the auc value:
airlines_gbm.auc(valid=True)
```

Lift Top Group

This parameter specifies that a model must improve its lift within the top 1% of the training data. To calculate the lift, H2O sorts each observation from highest to lowest predicted value. The top group or top 1% corresponds to the observations with the highest predicted values. Lift is the ratio of correctly classified positive observations (rows with a positive target) to the total number of positive observations within a group

Examples:

R | Python

```
# import H2OGradientBoostingEstimator and the airlines dataset:
from h2o.estimators import H2OGradientBoostingEstimator
airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/airlines/allyears2k_headers.zip")

# set the factors:
airlines["Year"]= airlines["Year"].asfactor()
airlines["Month"]= airlines["Month"].asfactor()
airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
airlines["Cancelled"] = airlines["Cancelled"].asfactor()
airlines['FlightNum'] = airlines['FlightNum'].asfactor()

# set the predictors and response columns:
predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
              "DayOfWeek", "Month", "Distance", "FlightNum"]
response = "IsDepDelayed"

# split the data into training and validation sets:
train, valid= airlines.split_frame(ratios = [.8], seed = 1234)

# build and train the model using the lifttopgroup stopping metric:
airlines_gbm = H2OGradientBoostingEstimator(stopping_metric = "lifttopgroup",
                                            stopping_rounds = 3,
                                            stopping_tolerance = 1e-2,
                                            seed = 1234)
airlines_gbm.train(x = predictors, y = response,
                   training_frame = train, validation_frame = valid)

# retrieve the auc value:
airlines_gbm.auc(valid = True)
```

Deviance

The model will stop building if the deviance fails to continue to improve. Deviance is computed as follows:

```
Loss = Quadratic -> MSE==Deviance For Absolute/Laplace or Huber -> MSE != Deviance
```

Examples:

R | **Python**

```
# import H2OGradientBoostingEstimator and the cars dataset:
from h2o.estimators import H2OGradientBoostingEstimator
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the predictors and response columns:
predictors = ["economy","cylinders","displacement","power","weight"]
response = "acceleration"

# split the data into training and validation sets:
train, valid = cars.split_frame(ratios = [.8],seed = 1234)

# build and train the model using the deviance stopping metric:
cars_gbm = H2OGradientBoostingEstimator(stopping_metric = "deviance",
                                        stopping_rounds = 3,
                                        stopping_tolerance = 1e-2,
                                        seed = 1234)
cars_gbm.train(x = predictors, y = response,
               training_frame = train, validation_frame = valid)

# retrieve the mse value:
cars_gbm.mse(valid = True)
```

Mean-Per-Class-Error

The model will stop building after the mean-per-class error rate fails to improve.

Examples:

R | **Python**

```
# import H2OGradientBoostingEstimator and the cars dataset:
from h2o.estimators import H2OGradientBoostingEstimator
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the predictors and response columns:
predictors = ["economy","cylinders","displacement","power","weight"]
response = "acceleration"

# split the data into training and validation sets:
train, valid = cars.split_frame(ratios=[.8],seed=1234)

# build and train the model using the meanperclasserror stopping metric:
cars_gbm = H2OGradientBoostingEstimator(stopping_metric = "meanperclasserror",
                                        stopping_rounds = 3,
                                        stopping_tolerance = 1e-2,
                                        seed = 1234)
cars_gbm.train(x=predictors, y=repsonse,
               training_frame=train, validation_frame=valid)

# retrieve the mse value:
cars_gbm.mse(valid = True)
```

In addition to the above options, Logloss, MSE, RMSE, MAE, RMSLE, and AUC can also be used as the stopping metric.

Model Performance Graphs

Confusion Matrix

A confusion matrix is a table depicting performance of algorithm in terms of false positives, false negatives, true positives, and true negatives. In H2O, the actual results display in the columns and the predictions display in the rows; correct predictions are highlighted in yellow. In the example below, `0` was predicted correctly 902 times, while `8` was predicted correctly 822 times and `0` was predicted as `4` once.

▾ TRAINING METRICS - CONFUSION MATRIX

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	902	0	10	5	1	12	3	3	7	3	0.0465	44 / 946
1	0	1057	8	4	2	6	4	6	14	7	0.0460	51 / 1,108
2	14	11	826	25	23	5	17	17	25	4	0.1458	141 / 967
3	7	6	15	900	2	39	2	16	33	11	0.1271	131 / 1,031
4	1	3	13	1	893	3	5	7	3	52	0.0897	88 / 981
5	14	7	7	25	13	814	29	3	26	15	0.1459	139 / 953
6	8	5	25	1	21	18	875	3	7	4	0.0951	92 / 967
7	6	10	10	9	9	1	0	893	0	44	0.0906	89 / 982
8	9	21	8	24	13	42	10	5	822	31	0.1655	163 / 985
9	7	6	4	6	40	5	0	39	11	885	0.1176	118 / 1,003
Total	968	1126	926	1000	1017	945	945	992	948	1056	0.1064	1,056 / 9,923

The class labels calculations vary based on whether this is a binary or multiclass classification problem.

- **Binary Classification**: All predicted probabilities greater than or equal to the F1 Max threshold are labeled with the positive class (e.g., 1, True, or the second label in lexicographical order). The F1 Max threshold is selected to maximize the F1 score calculated from confusion matrix values (true positives, true negatives, false positives, and false negatives).
- **Multiclass Classification**: Prediction class labels are based on the class with the highest predicted probability.

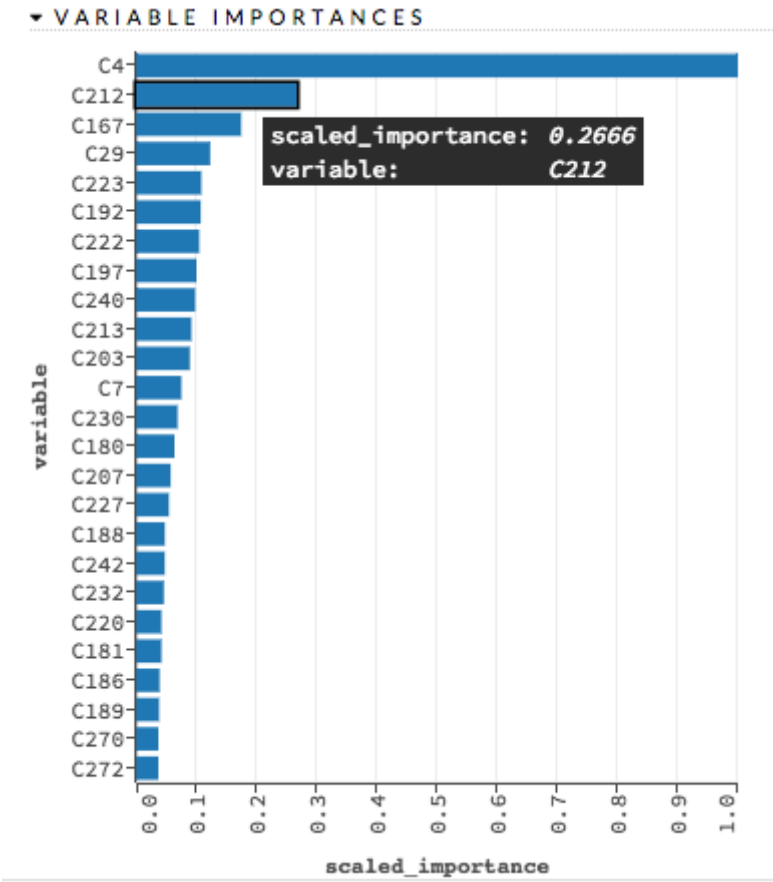
Examples:

R | **Python**

```
# import H2OGradientBoostingEstimator and the cars dataset:
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the factor:
cars["cylinders"] = cars["cylinders"].asfactor()

# split the data into training and validation sets:
train, valid = cars.split_frame(ratios=[.8],seed=1234)

# set the predictors columns, response column, and distribution type:
predictors = ["displacement", "power", "weight", "acceleration", "year"]
response_col = "cylinders"
distribution = "multinomial"

# build and train the model:
gbm = H2OGradientBoostingEstimator(nfolds = 3, distribution = distribution)
gbm.train(x=predictors, y=response_col, training_frame=train, validation_frame=valid)

# build the confusion matrix:
gbm.confusion_matrix(train)
```

Variable Importances

Variable importances represent the statistical significance of each variable in the data in terms of its affect on the model. Variables are listed in order of most to least importance. The percentage values represent the percentage of importance across all variables, scaled to 100%. The method of computing each variable's importance depends on the algorithm. More information is available in the Variable Importance section.



Examples:

R | **Python**

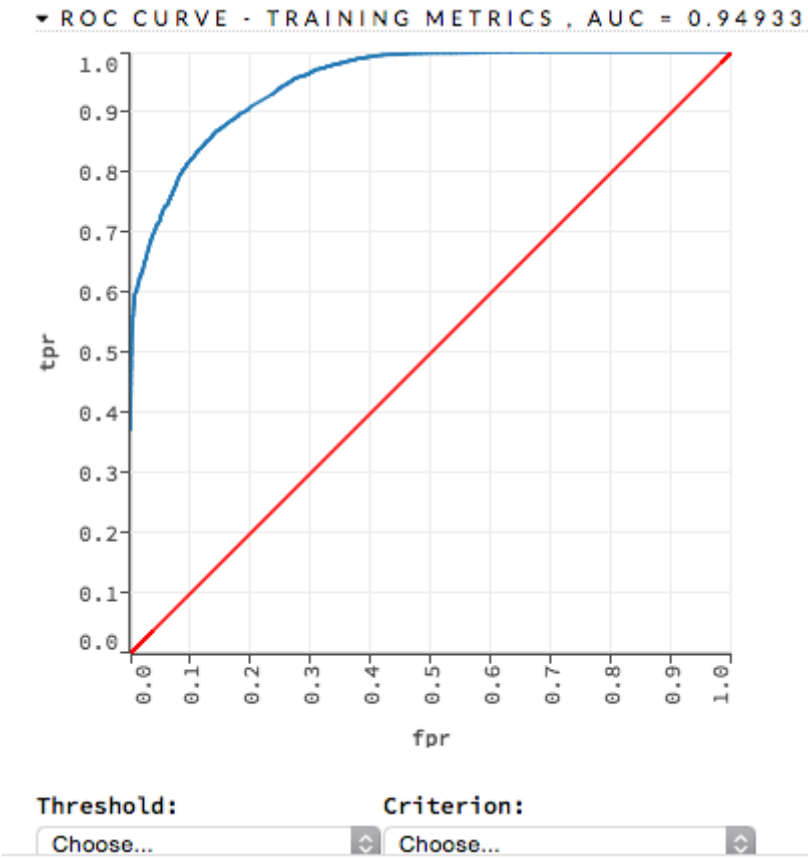
```
# import H2OGradientBoostingEstimator and the prostate dataset:
from h2o.estimators import H2OGradientBoostingEstimator
pros = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/prostate/prostate.csv.zip")

# set the factors:
pros[1] = pros[1].asfactor()
pros[3] = pros[3].asfactor()
pros[4] = pros[4].asfactor()
pros[5] = pros[5].asfactor()
pros[8] = pros[8].asfactor()

# split the data into training and validation sets:
train, valid = pros.split_frame(ratios=[.8], seed=1234)

# set the predictors and response columns:
predictors = ["AGE","RACE","DPROS","DCAPS","PSA","VOL","GLEASON"]
response = "CAPSULE"

# build and train the model:
pros_gbm = H2OGradientBoostingEstimator(nfolds=2)
pros_gbm.train(x = predictors, y = response,
               training_frame = train,
               validation_frame = valid)

# build the variable importances plot:
pros_gbm.varimp_plot()
```

ROC Curve

A ROC Curve is a graph that represents the ratio of true positives to false positives. (For more information, refer to the Linear Digressions podcast describing ROC Curves.) To view a specific threshold, select a value from the drop-down **Threshold** list. To view any of the following details, select it from the drop-down **Criterion** list:

- Max f1
- Max f2
- Max f0point5
- Max accuracy
- Max precision
- Max absolute MCC (the threshold that maximizes the absolute Matthew's Correlation Coefficient)
- Max min per class accuracy

The lower-left side of the graph represents less tolerance for false positives while the upper-right represents more tolerance for false positives. Ideally, a highly accurate ROC resembles the following example.



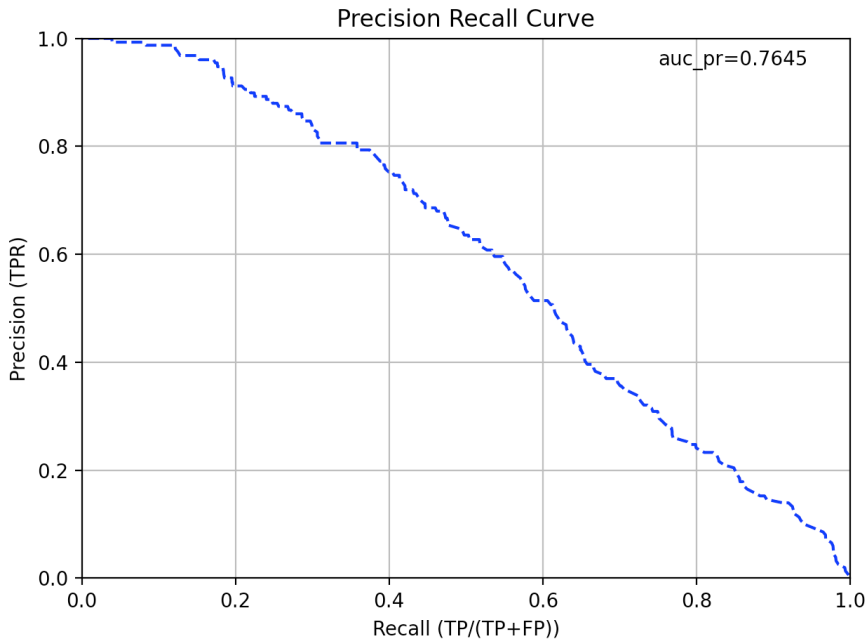
Examples:

R | **Python**

```
# import H2OGradientBoostingEstimator and the prostate dataset:
from h2o.estimators import H2OGradientBoostingEstimator
pros = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/prostate/prostate.csv.zip")

# set the factors:
pros[1] = pros[1].asfactor()
pros[3] = pros[3].asfactor()
pros[4] = pros[4].asfactor()
pros[5] = pros[5].asfactor()
pros[8] = pros[8].asfactor()

# set the predictors and response columns:
predictors = ["AGE","RACE","DPROS","DCAPS","PSA","VOL","GLEASON"]
response = "CAPSULE"

# split the data into training and validation sets:
train, test = pros.split_frame(ratios=[.8], seed=1234)

# build and train the model:
pros_gbm = H2OGradientBoostingEstimator(nfolds=2)
pros_gbm.train(x = predictors, y = response, training_frame = train)

# build the roc curve:
perf = pros_gbm.model_performance(test)
perf.plot(type = "roc")
```

AUCPR Curve

The area under the precision-recall curve graph represents how well a binary classification model is able to distinguish between precision recall pairs or points. The AUCPR does not care about True Negatives and is much more sensitive to True Positives, False Positives, and False Negatives than AUC.



Examples:

R | **Python**

```
# import H2OGeneralizedLinearEstimator and the prostate dataset:
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
pros = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/prostate/prostate.csv.zip")

# set the factors:
pros[1] = pros[1].asfactor()
pros[3] = pros[3].asfactor()
pros[4] = pros[4].asfactor()
pros[5] = pros[5].asfactor()
pros[8] = pros[8].asfactor()

# set the predictors and response column:
predictors = ["AGE","RACE","DPROS","DCAPS","PSA","VOL","GLEASON"]
response = "CAPSULE"

# split the data into training and validation sets:
train, valid = pros.split_frame(ratios=[.8], seed=1234)

# build and train the model:
glm_model = H2OGeneralizedLinearEstimator(family= "binomial",
                                          lambda_ = 0,
                                          compute_p_values = True)
glm_model.train(predictors, response, training_frame=train, validation_frame=valid)

# build the precision recall curve:
perf = glm_model.model_performance(valid)
perf.plot(type = "pr")
```

Hit Ratio

The hit ratio is a table representing the number of times that the prediction was correct out of the total number of predictions.

▾ OUTPUT - TRAINING_METRICS - TOP-10 HIT RATIOS

k	hit_ratio
1	0.9975
2	0.9998
3	1.0
4	1.0
5	1.0
6	1.0
7	1.0
8	1.0
9	1.0
10	1.0

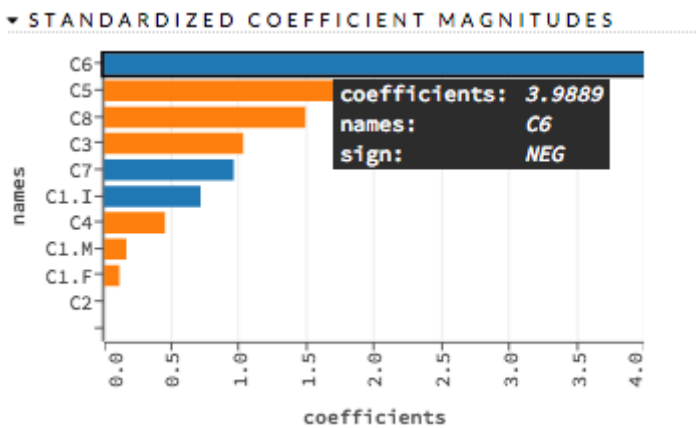
Examples:

R | Python

```
# import H2OGradientBoostingEstimator and the cars dataset:
from h2o.estimators import H2OGradientBoostingEstimator
cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/junit/cars_20mpg.csv")

# set the factor:
cars["cylinders"] = cars["cylinders"].asfactor()

# split the data into training and validation sets:
train, valid = cars.split_frame(ratios = [.8], seed = 1234)

# set the predictors columns, repsonse column, and distribution type:
predictors = ["displacement", "power", "weight", "acceleration", "year"]
response_col = "cylinders"
distribution = "multinomial"

# build and train the model:
gbm = H2OGradientBoostingEstimator(nfolds = 3, distribution = distribution)
gbm.train(x=predictors, y=response_col, training_frame=train, validation_frame=valid)

# build the hit ratio table:
gbm_hit = gbm.hit_ratio_table(valid=True)
gbm_hit.show()
```

Standardized Coefficient Magnitudes

This chart represents the relationship of a specific feature to the response variable. Coefficients can be positive (orange) or negative (blue). A positive coefficient indicates a positive relationship between the feature and the response, where an increase in the feature corresponds with an increase in the response, while a negative coefficient represents a negative relationship between the feature and the response where an increase in the feature corresponds with a decrease in the response (or vice versa).



Examples:

R | Python

```
# import H2OGeneralizedLinearEstimator and the prostate dataset:
from h2o.estimators import H2OGeneralizedLinearEstimator
pros = h2o.import_file("http://s3.amazonaws.com/h2o-public-test-
data/smalldata/prostate/prostate.csv.zip")

# set the factors:
pros[1] = pros[1].asfactor()
pros[3] = pros[3].asfactor()
pros[4] = pros[4].asfactor()
pros[5] = pros[5].asfactor()
pros[8] = pros[8].asfactor()

# set the predictors and response columns:
response = "CAPSULE"
predictors = ["AGE","RACE","PSA","DCAPS"]

# build and train the model:
glm = H2OGeneralizedLinearEstimator(nfolds = 5,
                                    alpha = 0.5,
                                    lambda_search = False,
                                    family = "binomial")
glm.train(x = predictors, y = response, training_frame = pros)

# build the standardized coefficient magnitudes plot:
glm.std_coef_plot()
```

Partial Dependence Plots

Use `partialPlot` (R)/`partial_plot` (Python) to create a partial dependece plot. This plot provides a graphical representation of the marginal effect of a variable on the class probability (binary and multiclass classification) or response (regression). Note that this is only available for models that include only numerical values.

The partial dependence of a given feature X_j is the average of the response function g, where all the components of X_j are set to x_j ($X_j = [x_j^{(0)}, \ldots, x_j^{(N-1)}]^T$)

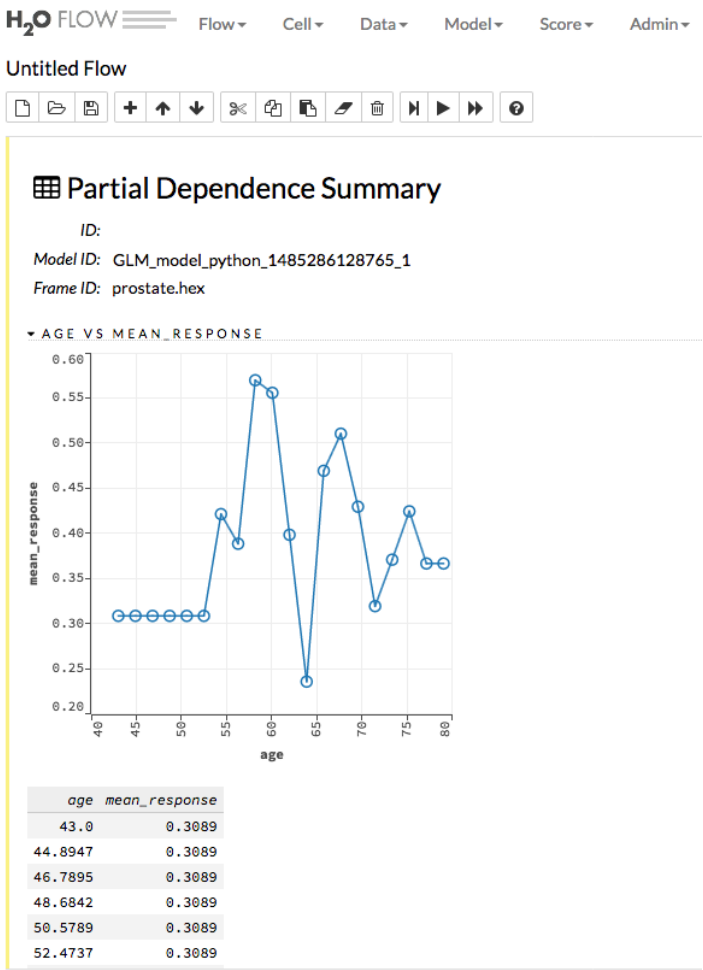
Thus, the one-dimensional partial dependence of function g on X_j is the marginal expectation:

$$PD(X_j, g) = E_{X_{(-j)}}\left[g(X_j, X_{(-j)})\right] = \frac{1}{N}\sum_{i=0}^{N-1} g(x_j, \mathbf{x}_{(-j)}^{(i)})$$

Notes:

- The partial dependence of a given feature is Xj (where j is the column index).
- You can also change the equation to sum from 1 to N instead of 0 to N-1.

- Instead of `cols`, you can use the `col_pairs_2dpdp` option along with a list containing pairs of column names to generate 2D partial dependence plots.
- Multiclass problems require an additional `targets` parameter. A Python demo is available showing how to retrieve PDPs for multiclass problems.



Defining a Partial Dependence Plot

The following can be specified when building a partial dependence plot.

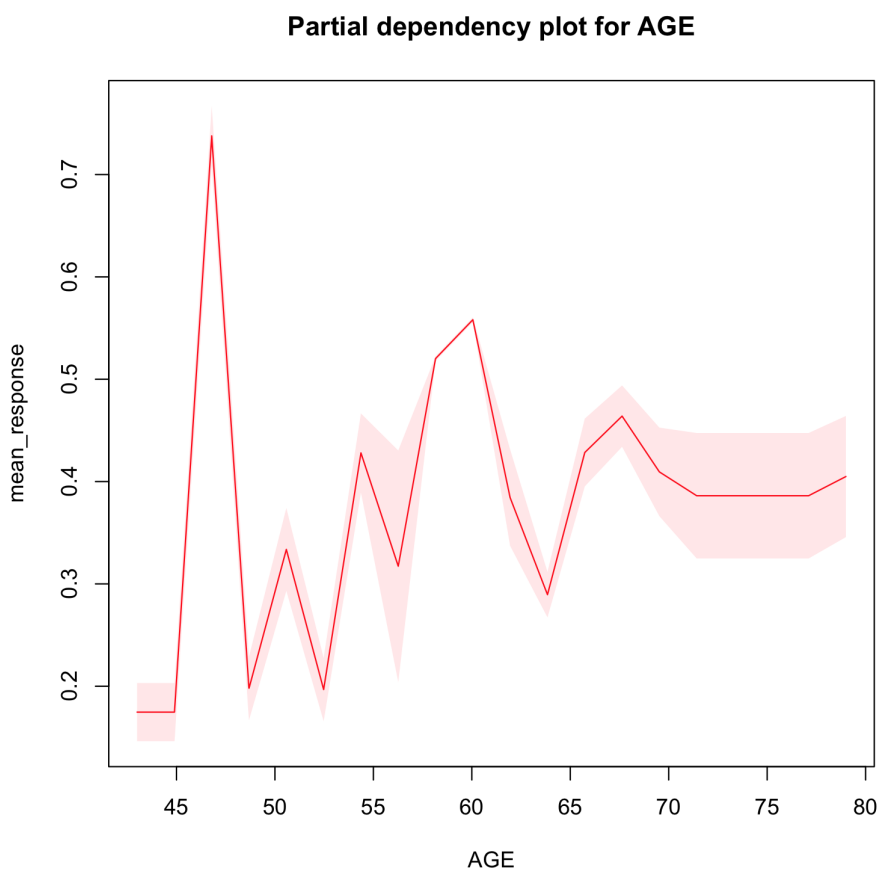
- `object`: (Required, R only) An H2OModel object.
- `data`: (Required) An H2OFrame object used for scoring and constructing the plot.
- `cols`: The feature(s) for which partial dependence will be calculated. One of either `col_pairs_2dpdp` or `cols` must be specified.
- `col_pairs_2dpdp`: A two-level nested list like this: col_pairs_2dpdp = list(c("col1_name", "col2_name"), c("col1_name","col3_name"), ...,) where a 2D partial plots will be generated for col1_name, col2_name pair, for col1_name, col3_name pair and whatever other pairs that are specified in the nested list. One of either `col_pairs_2dpdp` or `cols` must be specified.
- `destination_key`: A key reference to the created partial dependence tables in H2O.
- `nbins`: The number of bins used. For categorical columns make sure the number of bins exceed the level count. If you enable `include_na`, then the returned length will be nbins+1.

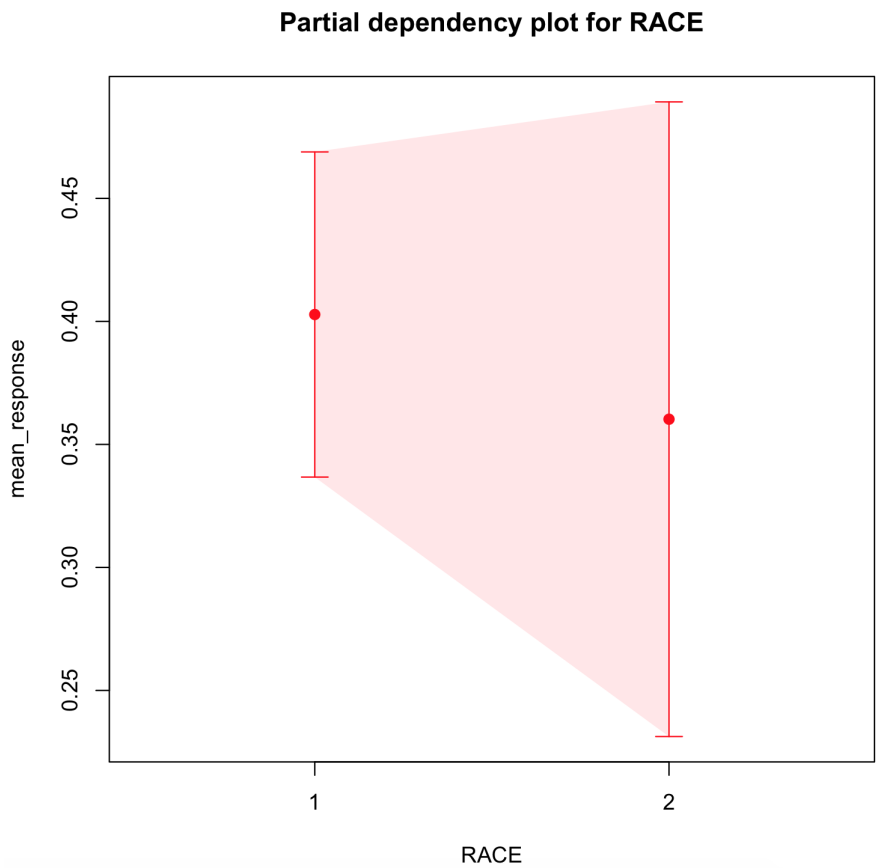
- `weight_column`: A string denoting which column of data should be used as the weight column.
- `plot`: A boolean specifying whether to plot partial dependence table.
- `plot_stddev`: A boolean specifying whether to add standard error to partial dependence plot.
- `figsize`: Specify the dimension/size of the returning plots. Adjust to fit your output cells.
- `server`: Specify whether to activate matplotlib "server" mode. In this case, the plots are saved to a file instead of being rendered.
- `include_na`: A boolean specifying whether missing value should be included in the Feature values.
- `user_splits`: A two-level nested list containing user-defined split points for pdp plots for each column. If there are two columns using user-defined split points, there should be two lists in the nested list. Inside each list, the first element is the column name followed by values defined by the user.
- `save_to` (R)/`save_to_file` (Python): Specify a fully qualified name to an image file that the resulting plot should be saved to, e.g. **/home/user/pdpplot.png**. The **png** postfix can be omitted. If the file already exists, it will be overridden. Note that you must also specify `plot = True` in order to save plots to a file.
- `row_index`: The row for which partial dependence will be calculated instead of the whole input frame.
- `targets`: (Required, multiclass only) Specify an array of one or more target classes when building PDPs for multiclass models. If you specify more than one class, then all classes are plot in one graph. (Note that in Flow, only one target can be specified.)

Binomial Examples

R | **Python**

```
# import H2OGradiantBoostingEstimator and the prostate dataset:
from h2o.estimators import H2OGradientBoostingEstimator
pros = h2o.import_file("http://s3.amazonaws.com/h2o-public-test-
data/smalldata/prostate/prostate.csv.zip")

# set the factors:
pros["CAPSULE"] = pros["CAPSULE"].asfactor()
pros["RACE"] = pros["RACE"].asfactor()

# set the predictors and response columns:
predictors = ["AGE","RACE"]
response = "CAPSULE"

# build and train the model:
pros_gbm = H2OGradientBoostingEstimator(ntrees = 10,
                                        max_depth = 5,
                                        learn_rate = 0.1,
                                        seed = 1234)
pros_gbm.train(x = predictors, y = response, training_frame = pros)

# build a 1-dimensional partial dependence plot:
pros_gbm.partial_plot(data = pros, cols = ["AGE","RACE"], server=True, plot = True)

# build a 2-dimensional partial dependence plot:
pdp2dOnly = pros_gbm.partial_plot(data = pros,
                                  server = True,
                                  plot = False,
                                  col_pairs_2dpdp = [['AGE', 'PSA'],['AGE', 'RACE']])
```

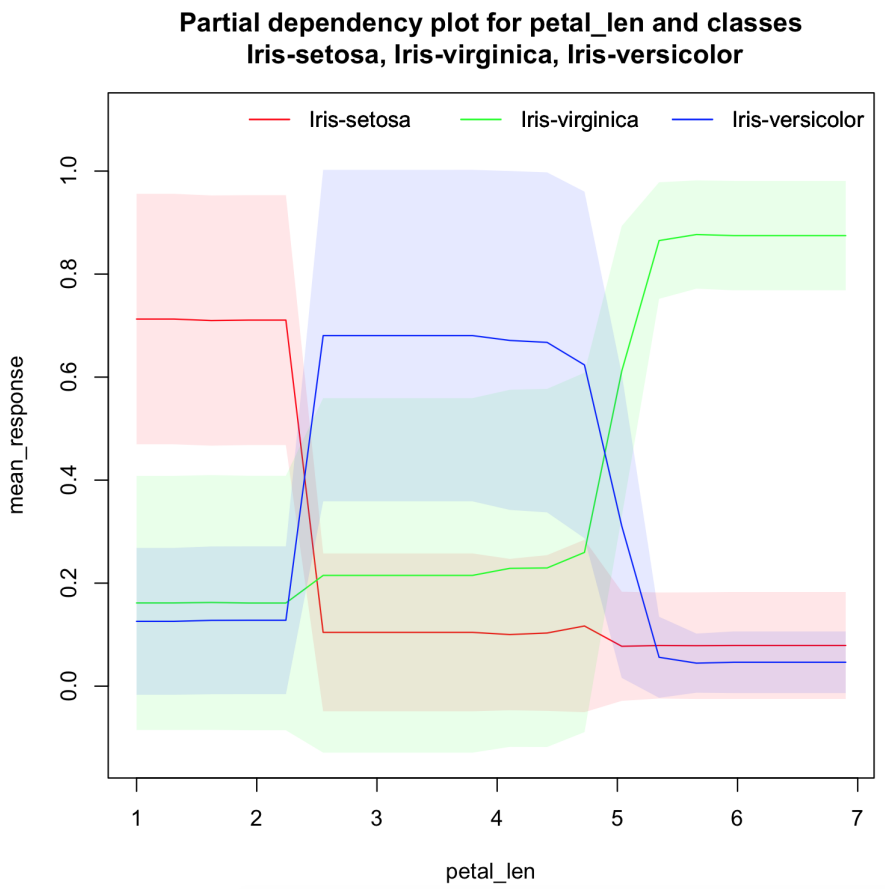




Multinomial Examples

R | Python

```
# import H2OGeneralizedLinearEstimator and the iris dataset:
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
iris = h2o.import_file("http://h2o-public-test-
data.s3.amazonaws.com/smalldata/iris/iris_wheader.csv")

# set the factors:
iris['class'] = iris['class'].asfactor()

# set the predictors and response columns:
predictors = iris.col_names[:-1]
response = 'class'

# split the dataset into train and valid sets:
train, valid = iris.split_frame(ratios = [.8], seed = 1234)

# build and train the model:
model = H2OGeneralizedLinearEstimator(family = 'multinomial', seed = 1234)
model.train(x = predictors,
            y = response,
            training_frame = train,
            validation_frame = valid)

# build the partial dependence plot:
cols = ["petal_len"]
targets = ["Iris-setosa", "Iris-virginica", "Iris-versicolor"]
pdp_petal_len_se = model.partial_plot(data = iris,
                                      cols = cols,
                                      targets = targets,
                                      plot_stddev = False,
                                      plot = True,
                                      server = True)
pdp_petal_len_se
```



Prediction

With H2O-3, you can generate predictions for a model based on samples in a test set using `h2o.predict()` or `predict()`. This can be accomplished in memory or using MOJOs/POJOs.

Note: MOJO/POJO predict cannot parse columns enclosed in double quotes (for example, ""2"").

For classification problems, predicted probabilities and labels are compared against known results. (Note that for binary models, labels are based on the maximum F1 threshold from the model object.) For regression problems, predicted regression targets are compared against testing targets and typical error metrics.

In-Memory Prediction

This section provides examples of performing predictions in Python and R. Refer to the Predictions topic in the Flow chapter to view an example of how to predict in Flow.

R | **Python**

```
import h2o
from h2o.estimators.gbm import H2OGradientBoostingEstimator
h2o.init()

# Import the prostate dataset
pros =
h2o.import_file("https://raw.github.com/h2oai/h2o/master/smalldata/logreg/prostate.csv")

# Convert the response column to a factor
pros["CAPSULE"] = pros["CAPSULE"].asfactor()

# Split the data into train and test
train, test = pros.split_frame(ratios = [.75], seed = 1234)

# Generate a GBM model using the training dataset
model = H2OGradientBoostingEstimator(distribution = "bernoulli",
                                     ntrees = 100,
                                     max_depth = 4,
                                     learn_rate = 0.1,
                                     seed = 1234)
model.train(y = "CAPSULE",
            x = ["AGE", "RACE", "PSA", "GLEASON"],
            training_frame = train)

# Predict using the GBM model and the testing dataset
predict = model.predict(test)

# View a summary of the prediction
predict
  predict         p0         p1
---------  ---------  ---------
        0  0.726079   0.273921
        0  0.81936    0.18064
        1  0.340055   0.659945
        0  0.985305   0.0146955
        0  0.941166   0.058834
        0  0.739246   0.260754
        1  0.431422   0.568578
        1  0.0971752  0.902825
        1  0.595788   0.404212
        1  0.639923   0.360077

[90 rows x 3 columns]
```

Predicting Leaf Node Assignment

For tree-based models, the `h2o.predict_leaf_node_assignment()` function predicts the leaf node assignment on an H2O model.

This function predicts against a test frame. For every row in the test frame, this function returns the leaf placements of the row in all the trees in the model. An optional Type can also be specified to define the placements. Placements can be represented either by paths to the

leaf nodes from the tree root (`Path` - default) or by H2O's internal identifiers (`Node_ID`). The order of the rows in the results is the same as the order in which the data was loaded.

This function returns an H2OFrame object with categorical leaf assignment identifiers for each tree in the model.

Using the previous example, run the following to predict the leaf node assignments:

R | **Python**

```
# Predict the leaf node assignment using the GBM model and test data.
# Predict based on the path from the root node of the tree.
predict_lna = model.predict_leaf_node_assignment(test, "Path")
```

Note: Leaf node assignment only works for trees of `max_depth` up to 63. For deeper trees, "NA" will be returned for paths of length 64 or more (-1 for node IDs).

Predict Contributions

In H2O-3, each returned H2OFrame has a specific shape (#rows, #features + 1). This includes a feature contribution column for each input feature, with the last column being the model bias (same value for each row). The sum of the feature contributions and the bias term is equal to the raw prediction of the model. Raw prediction of tree-based model is the sum of the predictions of the individual trees before the inverse link function is applied to get the actual prediction. For Gaussian distribution, the sum of the contributions is equal to the model prediction.

H2O-3 supports TreeSHAP for DRF, GBM, and XGBoost. For these problems, the `predict_contributions` returns a new H2OFrame with the predicted feature contributions - SHAP (SHapley Additive exPlanation) values on an H2O model. If you have SHAP installed, then graphical representations can be retrieved in Python using SHAP functions. (Note that retrieving graphs via R is not yet supported.) An .ipynb demo showing this example is also available here.

Note: Multinomial classification models are currently not supported.

R | **Python**

```
# Predict the contributions using the GBM model and test data.
contributions = model.predict_contributions(test)
contributions
```

```
       AGE        RACE         PSA    GLEASON    BiasTerm
----------  ----------  ----------  ---------  ----------
-0.946558   -0.474114    0.502114    0.187214   -0.243475
 0.522277   -0.354307   -0.733955   -0.702555   -0.243475
 1.16103     0.0383553   1.10714    -1.4        -0.243475
 0.0736866   0.0312205  -1.92878    -2.13806    -0.243475
-0.0201546   0.0658928  -1.59685    -0.977809   -0.243475
-0.334944   -0.581437   -0.409314    0.527115   -0.243475
 0.30024     0.0747187  -0.884867    1.02944    -0.243475
 0.373082    0.0791007   1.72772     0.292584   -0.243475
-0.0305803   0.047525   -1.05218     0.89077    -0.243475
 0.435522    0.0751563   0.0887201  -0.930953   -0.243475

[90 rows x 5 columns]
```

```
# Import required packages for running SHAP commands
import shap

# Load JS visualization code
shap.initjs()

# Convert the H2OFrame to use with SHAP's visualization functions
contributions_matrix = contributions.as_data_frame().as_matrix()

# Calculate SHAP values for all features
shap_values = contributions_matrix[:,0:4]

# Expected values is the last returned column
expected_value = contributions_matrix[:,4].min()

# Visualize the training set predictions
X=["AGE","RACE","PSA","GLEASON"]
shap.force_plot(expected_value, shap_values, X)

# Summarize the effects of all the features
shap.summary_plot(shap_values, X)

# View the same summary as a bar chart
shap.summary_plot(shap_values, X, plot_type="bar")
```

Predict Stage Probabilities

Use the `staged_predict_proba` function to predict class probabilities at each stage of an H2O Model. Note that this can only be used with GBM.

Using the previous example, run the following to predict probabilities at each stage in the model:

R | Python

```
# Predict the class probabilities using the GBM model and test data.
staged_predict_proba = model.staged_predict_proba(test)
```

Prediction Threshold

For classification problems, when running `h2o.predict()` or `.predict()`, the prediction threshold is selected as follows:

- If you only have training data, the max F1 threshold comes from the train data model.
- If you train a model with training and validation data, the max F1 threshold comes from the validation data model.
- If you train a model with training data and set the `nfolds` parameter, the Max F1 threshold from the training data model metrics is used.
- If you train a model with the train data and validation data and also set the `nfolds` parameter, the Max F1 threshold from the validation data model metrics is used.

Predict Feature Frequency

Use the `feature_frequencies` function to retrieve the number of times a feature was used on a prediction path in a tree model. This option is only available in GBM, DRF, and IF.

Using the previous example, run the following to the find frequency of each feature in the prediction path of the model:

R | Python

```
# Retrieve the number of occurrences of each feature for given observations
# on their respective paths in a tree ensemble model
feature_frequencies = model.feature_frequencies(train)
feature_frequencies
```

AGE	RACE	PSA	GLEASON
96	15	177	66
112	17	204	67
103	17	210	67
129	18	189	64
107	7	167	67
125	19	161	68
117	19	145	66
108	16	200	71
106	16	180	71
121	17	186	67

[290 rows x 4 columns]

Predict using MOJOs

An end-to-end example from building a model through predictions using MOJOs is available in the MOJO Quick Start topic.

Predict using POJOs

An end-to-end example from building a model through predictions using POJOs is available in the POJO Quick Start topic.