Generalized Linear Modeling with H2O

Tomas Nykodym Tom Kraljevic Nadine Hussami
Ariel Rao Amy Wang
Edited by: Jessica Lanford

http://h2o.ai/resources/

December 2016: Fifth Edition

Generalized Linear Modeling with H2O
by Tomas Nykodym, Tom Kraljevic,
Nadine Hussami, Ariel Rao, & Amy Wang
Edited by: Jessica Lanford

Published by H2O.ai, Inc.
2307 Leghorn St.
Mountain View, CA 94043

December 2016: Fifth Edition

Photos by ©H2O.ai, Inc.

Printed in the United States of America.

Contents

1 Introduction

This document introduces the reader to generalized linear modeling with H2O. Examples are written in R and Python. Topics include:

- installation of H2O
- basic GLM concepts
- building GLM models in H2O
- interpreting model output
- making predictions

2 What is H2O?

H2O is fast, scalable, open-source machine learning and deep learning for smarter applications. With H2O, enterprises like PayPal, Nielsen Catalina, Cisco, and others can use all their data without sampling to get accurate predictions faster. Advanced algorithms such as deep learning, boosting, and bagging ensembles are built-in to help application designers create smarter applications through elegant APIs. Some of our initial customers have built powerful domain-specific predictive engines for recommendations, customer churn, propensity to buy, dynamic pricing, and fraud detection for the insurance, healthcare, telecommunications, ad tech, retail, and payment systems industries.

Using in-memory compression, H2O handles billions of data rows in-memory, even with a small cluster. To make it easier for non-engineers to create complete analytic workflows, H2O's platform includes interfaces for R, Python, Scala, Java, JSON, and CoffeeScript/JavaScript, as well as a built-in web interface, Flow. H2O is designed to run in standalone mode, on Hadoop, or within a Spark Cluster, and typically deploys within minutes.

H2O includes many common machine learning algorithms, such as generalized linear modeling (linear regression, logistic regression, etc.), Naïve Bayes, principal components analysis, k-means clustering, and others. H2O also implements best-in-class algorithms at scale, such as distributed random forest, gradient boosting, and deep learning. Customers can build thousands of models and compare the results to get the best predictions.

H2O is nurturing a grassroots movement of physicists, mathematicians, and computer scientists to herald the new wave of discovery with data science by collaborating closely with academic researchers and industrial data scientists. Stanford university giants Stephen Boyd, Trevor Hastie, Rob Tibshirani advise

the H2O team on building scalable machine learning algorithms. With hundreds of meetups over the past three years, H2O has become a word-of-mouth phenomenon, growing amongst the data community by a hundred-fold, and is now used by 30,000+ users and is deployed using R, Python, Hadoop, and Spark in 2000+ corporations.

Try it out

- Download H2O directly at `http://h2o.ai/download`.
- Install H2O's R package from CRAN at `https://cran.r-project.org/web/packages/h2o/`.
- Install the Python package from PyPI at `https://pypi.python.org/pypi/h2o/`.

Join the community

- To learn about our meetups, training sessions, hackathons, and product updates, visit `http://h2o.ai`.
- Visit the open source community forum at `https://groups.google.com/d/forum/h2ostream`.
- Join the chat at `https://gitter.im/h2oai/h2o-3`.

3 Installation

H2O requires Java; if you do not already have Java installed, install it from `https://java.com/en/download/` before installing H2O.

The easiest way to directly install H2O is via an R or Python package.

3.1 Installation in R

To load a recent H2O package from CRAN, run:

```
1 install.packages("h2o")
```

Note: The version of H2O in CRAN may be one release behind the current version.

For the latest recommended version, download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in R" tab.
4. Copy and paste the commands into your R session.

After H2O is installed on your system, verify the installation:

```
library(h2o)

#Start H2O on your local machine using all available cores.
#By default, CRAN policies limit use to only 2 cores.
h2o.init(nthreads = -1)

#Get help
?h2o.glm
?h2o.gbm
?h2o.deeplearning

#Show a demo
demo(h2o.glm)
demo(h2o.gbm)
demo(h2o.deeplearning)
```

3.2 Installation in Python

To load a recent H2O package from PyPI, run:

```
pip install h2o
```

To download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in Python" tab.
4. Copy and paste the commands into your Python session.

After H2O is installed, verify the installation:

```
import h2o

# Start H2O on your local machine
h2o.init()

# Get help
help(h2o.estimators.glm.H2OGeneralizedLinearEstimator)
help(h2o.estimators.gbm.H2OGradientBoostingEstimator)
help(h2o.estimators.deeplearning.
    H2ODeepLearningEstimator)

# Show a demo
h2o.demo("glm")
h2o.demo("gbm")
h2o.demo("deeplearning")
```

3.3 Pointing to a Different H2O Cluster

The instructions in the previous sections create a one-node H2O cluster on your local machine.

To connect to an established H2O cluster (in a multi-node Hadoop environment, for example) specify the IP address and port number for the established cluster using the `ip` and `port` parameters in the `h2o.init()` command. The syntax for this function is identical for R and Python:

```
h2o.init(ip = "123.45.67.89", port = 54321)
```

3.4 Example Code

Python and R code for the examples in this document can be found here:

`https://github.com/h2oai/h2o-3/tree/master/h2o-docs/src/booklets/v2_2015/source/GLM_Vignette_code_examples`

The document source itself can be found here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/booklets/v2_2015/source/GLM_Vignette.tex`

3.5 Citation

To cite this booklet, use the following:

Nykodym, T., Kraljevic, T., Hussami, N., Rao, A., and Wang, A. (Dec 2016). *Generalized Linear Modeling with H2O*. `http://h2o.ai/resources/`.

4 Generalized Linear Models

Generalized linear models (GLMs) are an extension of traditional linear models. They have gained popularity in statistical data analysis due to:

- the flexibility of the model structure unifying the typical regression methods (such as linear regression and logistic regression for binary classification)
- the recent availability of model-fitting software
- the ability to scale well with large datasets

4.1 Model Components

The estimation of the model is obtained by maximizing the log-likelihood over the parameter vector β for the observed data

$$\max_{\beta} \text{ (GLM Log-likelihood)}.$$

In the familiar linear regression setup, the independent observations vector y is assumed to be related to its corresponding predictor vector x by

$$y = x^\top \beta + \beta_0 + \epsilon,$$

where β is the parameter vector, β_0 represents the intercept term and $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a gaussian random variable which is the noise in the model.

The response y is normally distributed $y \sim \mathcal{N}(x^\top \beta + \beta_0, \sigma^2)$ as well. Since it assumes additivity of the covariates, normality of the error term as well as constancy of the variance, this model is restrictive. Because these assumptions are not applicable to many problems and datasets, a more flexible model is beneficial.

GLMs relax the above assumptions by allowing the variance to vary as a function of the mean, non-normal errors and a non-linear relation between the response and covariates. More specifically, the response distribution is assumed to belong to the exponential family, which includes the Gaussian, Poisson, binomial,

multinomial and gamma distributions as special cases. The components of a GLM are:

- The random component f for the dependent variable y: the density function $f(y;\theta,\phi)$ has a probability distribution from the exponential family parametrized by θ and ϕ. This removes the restriction on the distribution of the error and allows for non-homogeneity of the variance with respect to the mean vector.
- The systematic component η: $\eta = X\beta$, where X is the matrix of all observation vectors x_i.
- The link function g: $E(y) = \mu = g^{-1}(\eta)$ which relates the expected value of the response μ to the linear component η. The link function can be any monotonic differentiable function. This relaxes the constraints on the additivity of the covariates, and it allows the response to belong to a restricted range of values depending on the chosen transformation g.

This generalization makes GLM suitable for a wider range of problems. An example of a particular case of the GLM representation is the familiar logistic regression model commonly used for binary classification in medical applications.

4.2 GLM in H2O

H2O's GLM algorithm fits generalized linear models to the data by maximizing the log-likelihood. The elastic net penalty can be used for parameter regularization. The model fitting computation is distributed, extremely fast, and scales extremely well for models with a limited number of predictors with non-zero coefficients ($\sim$ low thousands).

H2O's GLM fits the model by solving the following likelihood optimization with parameter regularization:

$$\max_{\beta,\beta_0} \text{ (GLM Log-likelihood } - \text{Regularization Penalty)}.$$

The elastic net regularization penalty is the weighted sum of the ℓ_1 and ℓ_2 norms of the coefficients vector. It is defined as

$$\lambda P_\alpha(\beta) = \lambda \left(\alpha \|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2 \right),$$

with no penalty on the intercept term. It is implemented by subtracting $\lambda P_\alpha(\beta)$ from the optimized likelihood. This induces sparsity in the solution and shrinks the coefficients by imposing a penalty on their size.

These properties are beneficial because they reduce the variance in the predictions and make the model more interpretable by selecting a subset of the given variables. For a specific α value, the algorithm can compute models for a single value of the tuning parameter λ or the full regularization path as in the glmnet package for R (refer to Regularization Paths for Generalized Linear Models via Coordinate Descent by Friedman et. al).

The elastic net parameter $\alpha \in [0, 1]$ controls the penalty distribution between the ℓ_1 (least absolute shrinkage and selection operator or lasso) and ℓ_2 (ridge regression) penalties. When $\alpha = 0$, the ℓ_1 penalty is not used and a ridge regression solution with shrunken coefficients is obtained. If $\alpha = 1$, the Lasso operator soft-thresholds the parameters by reducing all of them by a constant factor and truncating at zero. This sets a different number of coefficients to zero depending on the λ value.

H2O's GLM solves the following optimization over N observations:

$$\max_{\beta,\beta_0} \sum_{i=1}^{N} \log f\left(y_i; \beta, \beta_0\right) - \lambda \left(\alpha \|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2\right)$$

Similar to the methods discussed in Regularization Paths for Generalized Linear Models via Coordinate Descent by Friedman et. al, H2O can compute the full regularization path, starting from the null-model (evaluated at the smallest penalty $\lambda_{\max}$ for which all coefficients are set to zero) down to a minimally-penalized model.

To improve the efficiency of this search, H2O employs the strong rules as described in Strong Rules for Discarding Predictors in Lasso-type Problems by Bien et. al to filter out inactive columns (whose coefficients will remain equal to zero given the imposed penalty). Computing the full regularization path is useful for convergence because it uses warm starts for consecutive λ values, and gives an insight regarding the order in which the coefficients start entering the model.

Moreover, cross-validation can be used after fitting models for the full regularization path. H2O returns the optimal amount of regularization λ for the given problem and data by computing the errors on the validation dataset of the fitted models created using the training data.

4.3 Model Fitting

GLM models are fitted by finding the set of parameters that maximizes the likelihood of the data. For the Gaussian family, maximum likelihood consists of minimizing the mean squared error. This has an analytical solution and can be solved with a standard method of least squares.

This is also applicable when the ℓ_2 penalty is added to the optimization. For all other families and when the ℓ_1 penalty is included, the maximum likelihood problem has no analytical solution. An iterative method such as IRLSM, L-BFGS, the Newton method, or gradient descent, must be used. To select the solver, select the model and specify the exponential density.

4.4 Model Validation

After selecting the model, evaluate the precision of the estimates to determine its accuracy. The quality of the fitted model can be obtained by computing the goodness of fit between the predicted values that it generates and the given input data. Multiple measures of discrepancy may be used.

H2O returns the logarithm of the ratio of likelihoods, called deviance, and the Akaike information criterion (AIC) after fitting a GLM. A benchmark for a good model is the saturated or full model, which is the largest model that can be fitted. Assuming the dataset consists of N observations, the saturated model fits N parameters $\hat{\mu}_i$. Since it gives a model with one parameter per observation, its predictions trivially fit the data perfectly.

The deviance is the difference between the maximized log-likelihoods of the fitted and saturated models. Let $\ell(y;\hat{\mu})$ be the likelihood corresponding to the estimated means vector $\hat{\mu}$ from the maximization, and let $\ell(y;y)$ be the likelihood of the saturated model which is the maximum achievable likelihood.

The scaled deviance, which is defined as $D^*(y,\hat{\mu}) = 2(\ell(y;y) - \ell(y;\hat{\mu}))$, is used as a goodness of fit measure for GLMs. When the deviance obtained is too large, the model does not fit the data well.

Another metric to measure the quality of the fitted statistical model is the AIC, defined as $AIC = 2k - 2\log(\ell(y;\hat{\mu}))$, where k is the number of parameters included in the model and ℓ is the likelihood of the fitted model defined as above.

Given a set of models for a dataset, the AIC compares the qualities of the models with respect to one another. This provides a way to select the optimal one, which is the model with the lowest AIC score.

The AIC score does not give an absolute measure of the quality of a given model. It takes into account the number of parameters that are included in the model by increasing the penalty as the number of parameters increases. This prevents from obtaining a complex model that overfits the data, an aspect which is not considered in the deviance computation.

4.5 Regularization

This subsection discusses the effects of parameter regularization. Penalties are introduced to the model building process to avoid over-fitting, reduce variance of the prediction error, and handle correlated predictors. The two most common penalized models are ridge regression and Lasso (least absolute shrinkage and selection operator). The elastic net combines both penalties.

4.5.1 Lasso and Ridge Regression

Lasso represents the ℓ_1 penalty and is an alternative regularized least squares method that penalizes the sum of the absolute values of the coefficients $||\beta||_1 = \sum_{k=1}^{p} |\beta_k|$. Lasso leads to a sparse solution when the tuning parameter is sufficiently large. As the tuning parameter value λ is increased, all coefficients are set to zero. Since reducing parameters to zero removes them from the model, Lasso is a good selection tool.

Ridge regression penalizes the ℓ_2 norm of the model coefficients $\|\beta\|_2^2 = \sum_{k=1}^{p} \beta_k^2$. It provides greater numerical stability and is easier and faster to compute than Lasso. It keeps all the predictors in the model and shrinks them proportionally. Ridge regression reduces coefficient values simultaneously as the penalty is increased without however setting any of them to zero.

Variable selection is important in numerous modern applications with many covariates where the ℓ_1 penalty has proven to be successful. Therefore, if the number of variables is large or if the solution is known to be sparse, we recommend using Lasso, which will select a small number of variables for sufficiently high λ that could be crucial to the interpretability of the model. The ℓ_2 norm does not have this effect: it shrinks the coefficients, but does not set them exactly to zero.

The two penalties also differ in the presence of correlated predictors. The ℓ_2 penalty shrinks coefficients for correlated columns towards each other, while the ℓ_1 penalty tends to select only one of them and set the other coefficients to zero. Using the elastic net argument α combines these two behaviors.

The elastic net both selects variables and preserves the grouping effect (shrinking coefficients of correlated columns together). Moreover, while the number of predictors that can enter a Lasso model saturates at $\min(n, p)$ (where n is the number of observations and p is the number of variables in the model), the elastic net does not have this limitation and can fit models with a larger number of predictors.

4.5.2 Elastic Net Penalty

H2O supports elastic net regularization, which is a combination of the ℓ_1 and ℓ_2 penalties parametrized by the α and λ arguments (similar to Regularization Paths for Generalized Linear Models via Coordinate Descent by Friedman et. al).

- α controls the elastic net penalty distribution between the ℓ_1 and ℓ_2 norms. It can have any value in the $[0, 1]$ range or a vector of values (which triggers grid search). If $\alpha = 0$, H2O solves the GLM using ridge regression. If $\alpha = 1$, the Lasso penalty is used.
- λ controls the penalty strength. The range is any positive value or a vector of values (which triggers grid search). **Note:** Lambda values are capped at λ_{max}, which is the smallest λ for which the solution is all zeros (except for the intercept term).

The combination of the ℓ_1 and ℓ_2 penalties is beneficial, since the ℓ_1 induces sparsity while the ℓ_2 gives stability and encourages the grouping effect (where a group of correlated variables tends to be dropped or added into the model simultaneously). When focusing on sparsity, one possible use of the α argument involves using the ℓ_1 mainly with very little ℓ_2 penalty (α almost 1) to stabilize the computation and improve convergence speed.

4.6 GLM Model Families

The following subsection describes the GLM families supported in H2O.

4.6.1 Linear Regression (Gaussian Family)

Linear regression corresponds to the Gaussian family model: the link function g is the identity and the density f corresponds to a normal distribution. It is the simplest example of a GLM, but has many uses and several advantages over other families. For instance, it is faster and requires more stable computations.

It models the dependency between a response y and a covariates vector x as a linear function:

$$\hat{y} = x^T \beta + \beta_0.$$

The model is fitted by solving the least squares problem, which is equivalent to maximizing the likelihood for the Gaussian family:

$$\max_{\beta,\beta_0} -\frac{1}{2N} \sum_{i=1}^{N} (x_i^\top \beta + \beta_0 - y_i)^2 - \lambda \left(\alpha \|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2 \right).$$

The deviance is the sum of the squared prediction errors:

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

Included in the H2O package is a prostate cancer dataset. The data was collected by Dr. Donn Young at the Ohio State University Comprehensive Cancer Center for a study of patients with varying degrees of prostate cancer. The following example illustrates how to build a model to predict the volume (VOL) of tumors obtained from ultrasounds based on features such as age and race.

Example in R

```
1 library(h2o)
2 h2o.init()
3 path = system.file("extdata", "prostate.csv", package
     = "h2o")
4 h2o_df = h2o.importFile(path)
5 gaussian.fit = h2o.glm(y = "VOL", x = c("AGE", "RACE",
      "PSA", "GLEASON"), training_frame = h2o_df,
     family = "gaussian")
```

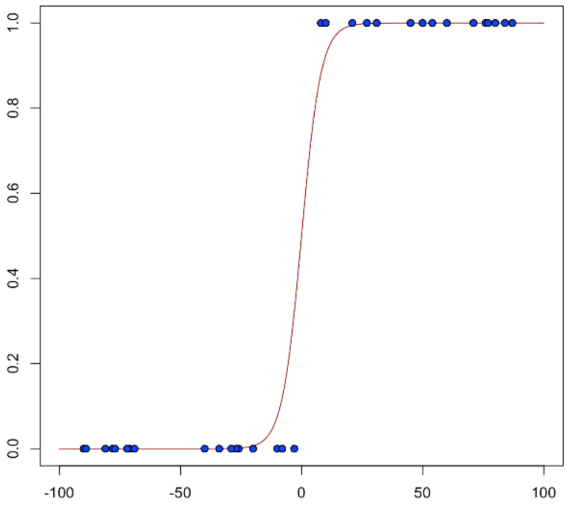
Example in Python

```
1 import h2o
2 from h2o.estimators.glm import
     H2OGeneralizedLinearEstimator
3 h2o.init()
```

```
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/prostate/prostate.csv")
gaussian_fit = H2OGeneralizedLinearEstimator(family =
    "gaussian")
gaussian_fit.train(y = "VOL", x = ["AGE", "RACE", "PSA
    ", "GLEASON"], training_frame = h2o_df)
```

4.6.2 Logistic Regression (Binomial Family)

Logistic regression is used for binary classification problems where the response is a categorical variable with two levels. It models the probability of an observation belonging to an output category given the data (for instance $Pr(y = 1|x)$). The canonical link for the binomial family is the logit function (also known as log odds). Its inverse is the logistic function, which takes any real number and projects it onto the $[0, 1]$ range as desired to model the probability of belonging to a class. The corresponding s-curve (or sigmoid function) is shown below,



and the fitted model has the form:

$$\hat{y} = Pr(y = 1|x) = \frac{e^{x^\top \beta + \beta_0}}{1 + e^{x^\top \beta + \beta_0}}$$

This can alternatively be written as:

$$\log\left(\frac{\hat{y}}{1-\hat{y}}\right) = \log\left(\frac{Pr(y=1|x)}{Pr(y=0|x)}\right) = x^\top\beta + \beta_0$$

The model is fitted by maximizing the following penalized likelihood:

$$\max_{\beta,\beta_0} \frac{1}{N}\sum_{i=1}^{N}\Big(y_i(x_i^\top\beta+\beta_0) - \log(1+e^{x_i^\top\beta+\beta_0})\Big) - \lambda\left(\alpha\|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2\right)$$

The corresponding deviance is equal to

$$D = -2\sum_{i=1}^{n}\left(y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right).$$

Using the prostate dataset, this example builds a binomial model that classifies the incidence of penetration of the prostatic capsule (CAPSULE). Confirm the entries in the CAPSULE column are binary using the `h2o.table()` function. Change the regression by changing the family to binomial.

Example in R

```
library(h2o)
h2o.init()
path = system.file("extdata", "prostate.csv", package
    = "h2o")
h2o_df = h2o.importFile(path)
is.factor(h2o_df$CAPSULE)
h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
is.factor(h2o_df$CAPSULE)
binomial.fit = h2o.glm(y = "CAPSULE", x = c("AGE", "
    RACE", "PSA", "GLEASON"), training_frame = h2o_df,
     family = "binomial")
```

4.6.3 Multi-class classification (Multinomial Family)

Multinomial family generalization of the binomial model is used for multi-class response variables. Similar to the binomial family, we model the conditional probability of observing class c given x. We have a vector of coefficients for each of the output classes (β is a matrix). The probabilities are defined as

$$\hat{y}_c = Pr(y = c|x) = \frac{e^{x^\top \beta_c + \beta_{c0}}}{\sum_{k=1}^{K}(e^{x^\top \beta_k + \beta_{k0}})}$$

The penalized negative log-likelihood is defined as:

$$-[\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}(y_{i,k}(x_i^\top \beta_k + \beta_{k0})) - log(\sum_{k=1}^{K} e^{x_i^\top \beta_k + \beta_{k0}})] + \lambda[\frac{(1-\alpha)}{2}\|\beta\|_F^2 + \alpha\sum_{j=1}^{P}\|\beta_j\|_1]$$

, where β_c is vector of coefficients for class c and $y_{i,k}$ is kth element of the binary vector produced by expanding the response variable using one-hot encoding (i.e. $y_{i,k} == 1$ iff the response at the ith observation is k. It is 0 otherwise.

Here is a simple example using the iris dataset:

Example in R

```
library(h2o)
h2o.init()
iris_h2o = as.h2o(iris)
h2o.fit = h2o.glm(training_frame=iris_h2o,y="Species",
    x=1:4,family="multinomial")
h2o.fit
```

Example in Python

```
import h2o
from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/iris/iris.csv")
multinomial_fit = H2OGeneralizedLinearEstimator(family
     = "multinomial")
multinomial_fit.train(y = 4, x = [0,1,2,3],
    training_frame = h2o_df)
```

4.6.4 Poisson Models

Poisson regression is typically used for datasets where the response represents counts and the errors are assumed to have a Poisson distribution. In general, it can be applied to any data where the response is non-negative. It models the dependency between the response and covariates as:

$$\hat{y} = e^{x^T \beta + \beta_0}$$

The model is fitted by maximizing the corresponding penalized likelihood:

$$\max_{\beta, \beta_0} \frac{1}{N} \sum_{i=1}^{N} \left(y_i (x_i^\top \beta + \beta_0) - e^{x_i^\top \beta + \beta_0} \right) - \lambda \left(\alpha \|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2 \right)$$

The corresponding deviance is equal to:

$$D = -2 \sum_{i=1}^{N} (y_i \log(y_i / \hat{y}_i) - (y_i - \hat{y}_i))$$

This example loads the Insurance data from the MASS library, imports it into H2O, and runs a Poisson model that predicts the number of claims (Claims) based on the district of the policy holder (District), their age (Age), and the type of car they own (Group).

Example in R

```
library(h2o)
h2o.init()
library(MASS)
data(Insurance)

# Convert ordered factors into unordered factors.
# H2O only handles unordered factors today.
class(Insurance$Group) <- "factor"
class(Insurance$Age) <- "factor"

# Copy the R data.frame to an H2OFrame using as.h2o()
h2o_df = as.h2o(Insurance)
```

```
poisson.fit = h2o.glm(y = "Claims", x = c("District",
    "Group", "Age"), training_frame = h2o_df, family =
     "poisson")
```

Example in Python

```
# Used swedish insurance data from smalldata instead
    of MASS/insurance due to the license of the MASS R
     package.
import h2o
from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
h2o.init()

h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/glm_test/
    Motor_insurance_sweden.txt", sep = '\t')
poisson_fit = H2OGeneralizedLinearEstimator(family = "
    poisson")
poisson_fit.train(y="Claims", x = ["Payment", "Insured
    ", "Kilometres", "Zone", "Bonus", "Make"],
    training_frame = h2o_df)
```

4.6.5 Gamma Models

The gamma distribution is useful for modeling a positive continuous response variable, where the conditional variance of the response grows with its mean but the coefficient of variation of the response $\sigma^2(y_i)/\mu_i$ is constant. It is usually used with the log link $g(\mu_i) = \log(\mu_i)$, or the inverse link $g(\mu_i) = \frac{1}{\mu_i}$ which is equivalent to the canonical link.

The model is fitted by solving the following likelihood maximization:

$$\max_{\beta,\beta_0} -\frac{1}{N}\sum_{i=1}^{N} \frac{y_i}{x_i^\top \beta + \beta_0} + \log(x_i^\top \beta + \beta_0) - \lambda\left(\alpha\|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2\right)$$

The corresponding deviance is equal to:

$$D = 2\sum_{i=1}^{N} -\log\left(\frac{y_i}{\hat{y}_i}\right) + \frac{(y_i - \hat{y}_i)}{\hat{y}_i}$$

To change the link function from the default inverse function to the log link function, modify the `link` argument.

Example in R

```
library(h2o)
h2o.init()
path = system.file("extdata", "prostate.csv", package
    = "h2o")
h2o_df = h2o.importFile(path)
gamma.inverse <- h2o.glm(y = "DPROS", x = c("AGE","
    RACE","CAPSULE","DCAPS","PSA","VOL"), training_
    frame = h2o_df, family = "gamma", link = "inverse"
    )
gamma.log <- h2o.glm(y="DPROS", x = c("AGE","RACE","
    CAPSULE","DCAPS","PSA","VOL"), training_frame =
    h2o_df, family = "gamma", link = "log")
```

Example in Python

```
import h2o
from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/prostate/prostate.csv")
gamma_inverse = H2OGeneralizedLinearEstimator(family =
     "gamma", link = "inverse")
gamma_inverse.train(y = "DPROS", x = ["AGE","RACE","
    CAPSULE","DCAPS","PSA","VOL"], training_frame =
    h2o_df)

gamma_log = H2OGeneralizedLinearEstimator(family = "
    gamma", link = "log")
gamma_log.train(y="DPROS", x = ["AGE","RACE","CAPSULE"
    ,"DCAPS","PSA","VOL"], training_frame = h2o_df)
```

4.6.6 Tweedie Models

Tweedie distributions are a family of distributions which include gamma, normal, Poisson and their combination. It is especially useful for modeling positive continuous variables with exact zeros. The variance of the Tweedie distribution is proportional to the p-th power of the mean $var(y_i) = \phi\mu_i^p$.

The Tweedie distribution is parametrized by variance power p. It is defined for all p values except in the $(0, 1)$ interval, and has the following distributions as special cases.

- $p = 0$: Normal
- $p = 1$: Poisson
- $p \in (1, 2)$: Compound Poisson, non-negative with mass at zero
- $p = 2$: Gamma
- $p = 3$: Inverse-Gaussian
- $p > 2$: Stable, with support on the positive reals

Example in R

```
library(h2o)
h2o.init()
library(HDtweedie)
data(auto) # 2812 policy samples with 56 predictors

dim(auto$x)
hist(auto$y)

# Copy the R data.frame to an H2OFrame using as.h2o()
h2o_df = as.h2o(auto)
vars= paste("x.",colnames(auto$x),sep="")
tweedie.fit = h2o.glm(y = "y", x = vars, training_
    frame = h2o_df, family = "tweedie")
```

Example in Python

```
import h2o
from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/glm_test/auto.csv")
tweedie_fit = H2OGeneralizedLinearEstimator(family = "
    tweedie")
tweedie_fit.train(y = "y", x = h2o_df.col_names[1:],
    training_frame = h2o_df)
```

The normal and Poisson examples have already been covered in the previous sections. For $p > 1$, the model likelihood to maximize has the form:

$$\max_{\beta,\beta_0} \sum_{i=1}^{N} \log(a(y_i,\phi)) + \left(\frac{1}{\phi}\left(\frac{y_i \mu_i^{1-p}}{1-p} - \kappa(\mu_i,p)\right)\right) - \lambda\left(\alpha\|\beta\|_1 + \frac{1}{2}(1-\alpha)\|\beta\|_2^2\right),$$

where $\kappa(\mu,p) = \mu^{2-p}/(2-p)$ for $p \neq 2$ and $\kappa(\mu,p) = \log(\mu)$ for $p = 2$ and where the function $a(y_i,\phi)$ is evaluated using series expansion, since it does not have an analytical solution. The link function in the GLM representation of the Tweedie distribution defaults to $g(\mu) = \mu^q = \eta = X\beta$ with $q = 1-p$. The

link power q can be set to other values, including $q = 0$ which is interpreted as $\log(\mu) = \eta$.

The corresponding deviance when $p \neq 1$ and $p \neq 2$ is equal to:

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

5 Building GLM Models in H2O

H2O's GLM implementation presents a high-performance distributed algorithm that scales linearly with the number of rows and works extremely well for datasets with a limited number of active predictors.

5.1 Classification and Regression

GLM can produce two categories of models: classification (binary classification only) and regression. Logistic regression is the GLM to perform binary classification.

The data type of the response column determines the model category. If the response is a categorical variable (also called a factor or an enum), then a classification model is created. If the response column data type is numeric (either integer or real), then a regression model is created.

The following examples show how to coerce the data type of a column to a factor.

Example in R

```
1 library(h2o)
2 h2o.init()
3 path = system.file("extdata", "prostate.csv", package
      = "h2o")
4 h2o_df = h2o.importFile(path)
5 h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
6 summary(h2o_df)
```

Example in Python

```
import h2o
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/prostate/prostate.csv")
h2o_df["CAPSULE"] = h2o_df["CAPSULE"].asfactor()
h2o_df.summary()
```

5.2 Training and Validation Frames

Frame refers to an H2OFrame, the fundamental method of data storage in H2O's distributed memory.

`training_frame` refers to a frame containing a training dataset. All predictors and the response (as well as offset and weights, if specified) must be included in this frame.

`validation_frame` refers to an optional frame containing a validation dataset. If specified, this frame must have exactly the same columns as the training dataset. Metrics are calculated on the validation dataset for convenience.

5.3 Predictor and Response Variables

Every model must specify its predictors and response. Predictors and responses are specified by the x and y parameters.

x contains the list of column names or column indices referring to vectors from the training frame; periods are not supported characters.

y is a column name or index referring to a vector from the training frame.

5.3.1 Categorical Variables

If the response column is categorical, then a classification model is created. GLM only supports binary classification, so the response column may only have two levels. Categorical predictor columns may have more than two levels.

We recommend letting GLM handle categorical columns, as it can take advantage of the categorical column for better performance and memory utilization.

We strongly recommend avoiding one-hot encoding categorical columns with many levels into many binary columns, as this is very inefficient. This is especially true for Python users who are used to expanding their categorical variables manually for other frameworks.

5.4 Family and Link

Family and Link are optional parameters. The default family is `Gaussian` and the default link is a canonical link for the selected family. These are passed as strings, e.g. `family = "gamma", link = "log"`. While it is possible to select a non-canonical link, this may lead to an unstable computation.

5.5 Regularization Parameters

To get the best possible model, we need to find the optimal values of the regularization parameters α and λ. To find the optimal values, H2O provides grid search over α and a special form of grid search called "lambda search" over λ. For a detailed explanation, refer to **Regularization**.

The recommended way to find optimal regularization settings on H2O is to do a grid search over a few α values with an automatic lambda search for each α. Both are described below in greater detail.

5.5.1 Alpha and Lambda

The `alpha` parameter controls the distribution between the ℓ_1 (Lasso) and ℓ_2 (Ridge regression) penalties. A value of 1.0 for `alpha` represents Lasso, and an `alpha` value of 0.0 produces ridge regression.

The `lambda` parameter controls the amount of regularization applied. If `lambda` is 0.0, no regularization is applied and the `alpha` parameter is ignored. The default value for `lambda` is calculated by H2O using a heuristic based on the training data. If you let H2O calculate the value for `lambda`, you can see the chosen value in the model output.

5.5.2 Lambda Search

Lambda search enables efficient and automatic search for the optimal value of the `lambda` parameter. When lambda search is enabled, GLM will first fit a model with maximum regularization and then keep decreasing it until overfitting occurs. The resulting model is based on the best `lambda` value.

When looking for sparse solution (`alpha` > 0), lambda search can also be used to efficiently handle very wide datasets because it can filter out inactive predictors (known as noise) and only build models for a small subset of predictors. A common use of lambda search is to run it on a dataset with many predictors but limit the number of active predictors to a relatively small value.

Lambda search can be enabled by setting `lambda_search` and can be configured using the following arguments:

- `alpha`: Regularization distribution between ℓ_1 and ℓ_2.
- `validation_frame` andor `n_folds`: Used to select the best lambda based on the cross-validation performance or the validation or training data. If available, cross-validation performance takes precedence. If no validation data is available, the best lambda is selected based on training data performance and is therefore guaranteed to always be the minimal lambda computed, since GLM can not overfit on a training dataset.

  Note: If running lambda search with a validation dataset and cross-validation disabled, the chosen lambda value corresponds to the lambda with the lowest validation error. The validation dataset is used to select the model and the model performance should be evaluated on another independent test dataset.

- `lambda_min_ratio` and `nlambdas`: The sequence of λs is automatically generated as an exponentially decreasing sequence. It ranges from λ_{max}, the smallest λ so that the solution is a model with all 0s, to $\lambda_{min} =$ *lambda_min_ratio* * λ_{max}.

  H2O computes λ-models sequentially and in decreasing order, warm-starting the model for λ_k with the solution for λ_{k-1}. By warm-starting (using the previous solution as the initial prediction) the models, we get better performance: typically models for subsequent λs are close to each other, so only a few iterations per λ are needed (typically two or three). This also achieves greater numerical stability, since models with a higher penalty are easier to compute. This method starts with an easy problem and then continues to make small adjustments.

  Note: `nlambda` and `lambda_min_ratio` also specify the relative distance of any two lambdas in the sequence. This is important when applying recursive strong rules, which are only effective if the neighboring lambdas are "close" to each other. The default values are `nlambda` $=$ 100 and $\lambda_{min} = \lambda_{max} 1e^{-4}$, which gives us the ratio of 0.912. For best results when using strong rules, keep the ratio close to the default.

- `max_active_predictors`: This limits the number of active predictors (the actual number of non-zero predictors in the model is going to be slightly lower). It is useful when obtaining a sparse solution to avoid costly computation of models with too many predictors.

5.6 Solver Selection

This section provides general guidelines for best performance from the H2O GLM implementation options. The optimal solver depends on the data properties and prior information regarding the variables (if available).

The data are considered sparse if the ratio of zeros to non-zeros in the input matrix is greater than ~ 10. The solution is sparse when only a subset of the original set of variables is intended to be kept in the model. In a dense solution, all predictors have non-zero coefficients in the final model.

5.6.1 Solver Details

H2O's GLM offers two different solvers:

- the Iteratively Reweighted Least Squares Method (IRLSM)
- the Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (L-BFGS)

IRLSM uses a Gram Matrix approach, which is very efficient for tall and narrow datasets and when running lambda search with a sparse solution. For wider and dense datasets (thousands of predictors and up), the L-BFGS solver scales better. If there are fewer than ~ 500 predictors in the data, use the default, which is IRLSM.

For larger numbers of predictors, it is recommended to run IRLSM with lambda search and compare it to L-BFGS with just the ℓ_2 penalty. For advanced users, we recommend the following general guidelines:

- For a dense solution and a dense dataset, use IRLSM if there are fewer than ~ 500 predictors in the data; otherwise, use L-BFGS. Set `alpha` to 0 to include ℓ_2 regularization in the elastic net penalty term to avoid inducing sparsity in the model.
- For a dense solution with a sparse dataset, use IRLSM if there are fewer than ~ 2000 predictors in the data; otherwise, use L-BFGS. Set `alpha` to 0.

- For a sparse solution with a dense dataset, use IRLSM with lambda-search if fewer than ~ 500 active predictors in the solution are expected; otherwise, use L-BFGS. Set `alpha` to be greater than zero to add an ℓ_1 penalty to the elastic net regularization, which induces sparsity in the estimated coefficients.
- For a sparse solution with a sparse dataset, use IRLSM with lambda-search if you expect less than ~ 5000 active predictors in the solution; otherwise, use L-BFGS. Set `alpha` to be greater than zero.
- If unsure whether the solution should be sparse or dense, try both and a grid of alpha values. The optimal model can be picked based on its performance on the validation data (or alternatively the performance in cross-validation when not enough data is available to have a separate validation dataset).

The above recommendations are general guidelines; if the performance of the method seems slow, experiment with the available options.

IRLSM can be run with two algorithms to solve its innermost loop: ADMM and cyclical coordinate descent. The latter is used in glmnet.

The method is able to handle large datasets well and deals efficiently with sparse features. It should improve the performance when the data contains categorical variables with a large number of levels, as it is implemented to deal with such variables in a parallelized way.

Coordinate descent can be implemented with naive or covariance updates as explained in the glmnet paper. The covariance updates version is faster when $N > p$ and $p \sim 500$.

In summary, the solver options for fitting a GLM in H2O are:

- **L-BFGS**: when the number of predictors is large
- **IRLSM**: IRLSM with the ADMM solver in the innermost loop when you have a limited number of predictors

5.6.2 Stopping Criteria

When using the ℓ_1 penalty with lambda search, specify a value for the `max_active_predictors` parameter to stop the search before it completes.

Models built at the beginning of the lambda search have higher lambda values, consider fewer predictors, and take less time to calculate the model.

Models built at the end of the lambda search have lower lambda values, incorporate more predictors, and take a longer time to calculate the model. Set the `nlambdas` parameter for a lambda search to specify the number of models attempted across the search.

Example in R

```
library(h2o)
h2o.init()
h2o_df = h2o.importFile("http://s3.amazonaws.com/h2o-public-test-data/smalldata/airlines/allyears2k_headers.zip")

#stops the model when we reach 10 active predictors
model = h2o.glm(y = "IsDepDelayed", x = c("Year", "Origin"), training_frame = h2o_df, family = "binomial", lambda_search = TRUE, max_active_predictors = 10)
print(model)
```

Example in Python

```
import h2o
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
h2o.init()
h2o_df = h2o.import_file("http://s3.amazonaws.com/h2o-public-test-data/smalldata/airlines/allyears2k_headers.zip")

#stops the model when we reach 10 active predictors
model = H2OGeneralizedLinearEstimator(family = "binomial", lambda_search = True, max_active_predictors = 10)
model.train(y = "IsDepDelayed", x = ["Year", "Origin"], training_frame = h2o_df)
print(model)
```

5.7 Advanced Features

H2O's GLM has several advanced features to help build better models.

5.7.1 Standardizing Data

The `standardize` parameter, which is enabled by default, standardizes numeric columns to have zero mean and unit variance. This parameter must be enabled (using `standardize=TRUE`) to produce standardized coefficient magnitudes in the model output.

We recommend enabling standardization when using regularization (i.e. `lambda` is chosen by H2O or greater than 0). Only advanced users should disable standardization.

5.7.2 Auto-remove collinear columns

Collinear columns can cause problems during model fitting. The preferred way to deal with collinearity is to add some regularization (either L1, L2 or Elastic Net). This is the default H2O behavior. However, if you want a non-regularized solution, you can choose to automatically remove collinear columns by setting the `remove_collinear_columns` option.

This option can only be used with the `IRLSM` solver and no regularization. If selected, H2O will automatically remove columns if it detects collinearity. Which columns are removed depends on the order of the columns in the vector of coefficients (Intercept first, then categorical variables ordered by cadrinality from largest to smallest, and then numbers).

Example in R

```
library(h2o)
h2o.init()
a = runif(100)
b = 2*a
c = -3*a + 10
df = data.frame(a,b,c)
h2o_df = as.h2o(df)
h2o.fit = h2o.glm(y = "c", x = c("a", "b"), training_
    frame = h2o_df, lambda=0,remove_collinear_columns=
    TRUE)
```

```
9 h2o.fit
```

5.7.3 P-Values

Z-score, standard error and p-values are classical statistical measures of model quality. p-values are essentially hypothesis tests on the values of each coefficient. A high p-value means that a coefficient is unreliable (insiginificant) while a low p-value suggest that the coefficient is statistically significant.

You can request p-values by setting the `compute_p_values` option. It can only be used with the `IRLSM` solver and no regularization. It is recommended that you also set the `remove_collinear_columns` option. Otherwise, H2O will return an error if it detects collinearity in the dataset and p-values are requested.

Note: GLM auto-standardizes the data by default (recommended). This changes the p-value of the constant term (intercept).

Example in R

```
1 library(h2o)
2 h2o.init()
3 a = runif(100)
4 b = runif(100)
5 c = -3*a + 10 + 0.01*runif(100)
6 df = data.frame(a,b,c)
7 h2o_df = as.h2o(df)
8 h2o.fit = h2o.glm(y = "c", x = c("a", "b"), training_
    frame = h2o_df, lambda=0,remove_collinear_columns=
    TRUE,compute_p_values=TRUE)
9 h2o.fit
```

5.7.4 K-fold Cross-Validation

All validation values can be computed using either the training dataset (the default option) or using K-fold cross-validation (`kfolds` > 1). When K-fold cross-validation is enabled, H2O randomly splits data into K equally-sized sections, trains each of the K models on $K-1$ sections, and computes validation on the section that was not used for training.

You can also specify the rows assigned to each fold using the `fold_assignment` or `fold_column` parameters.

Example in R

```
library(h2o)
h2o.init()
path = system.file("extdata", "prostate.csv", package = "h2o")
h2o_df = h2o.importFile(path)
h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
binomial.fit = h2o.glm(y = "CAPSULE", x = c("AGE", "RACE", "PSA", "GLEASON"), training_frame = h2o_df, family = "binomial", nfolds = 5)
print(binomial.fit)
print(paste("training auc:        ", binomial.fit@model$training_metrics@metrics$AUC))
print(paste("cross-validation auc:", binomial.fit@model$cross_validation_metrics@metrics$AUC))
```

Example in Python

```
import h2o
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.s3.amazonaws.com/smalldata/prostate/prostate.csv")
h2o_df['CAPSULE'] = h2o_df['CAPSULE'].asfactor()
binomial_fit = H2OGeneralizedLinearEstimator(family = "binomial", nfolds=5, fold_assignment="Random")
binomial_fit.train(y = "CAPSULE", x = ["AGE", "RACE", "PSA", "GLEASON"], training_frame = h2o_df)
print "training auc:", binomial_fit.auc(train=True)
print "cross-validation auc:", binomial_fit.auc(xval=True)
```

5.7.5 Grid Search Over Alpha

Alpha search is not always necessary; changing its value to 0.5 (or 0 or 1 if we only want Ridge or Lasso, respectively) works in most cases. If α search is required, specifying only a few values is typically sufficient. Alpha search is invoked by supplying a list of values for α instead of a single value. H2O then produces one model per α value.

The grid search computation can be done in parallel (depending on the cluster resources) and it is generally more efficient than computing different models separately from R.

Use caution when including $\alpha = 0$ or $\alpha = 1$ in the grid search. $\alpha = 0$ will produce a dense solution and it can be very slow (or even impossible) to compute in large N situations. $\alpha = 1$ has no ℓ_2 penalty, so it is therefore less numerically stable and can be very slow as well due to slower convergence. In general, we recommend using $alpha = 1 - \epsilon$ instead.

Example in R

```
1  library(h2o)
2  h2o.init()
3  path = system.file("extdata", "prostate.csv", package
       = "h2o")
4  h2o_df = h2o.importFile(path)
5  h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
6  alpha_opts = list(list(0), list(.25), list(.5), list
       (.75), list(1))
7  hyper_parameters = list(alpha = alpha_opts)
8  grid <- h2o.grid("glm", hyper_params = hyper_
       parameters,
9                   y = "CAPSULE", x = c("AGE", "RACE", "
                        PSA", "GLEASON"), training_frame
                        = h2o_df, family = "binomial")
10 grid_models <- lapply(grid@model_ids, function(model_
       id) { model = h2o.getModel(model_id) })
11 for (i in 1:length(grid_models)) {
12     print(sprintf("regularization: %-50s  auc: %f",
           grid_models[[i]]@model$model_summary$
           regularization, h2o.auc(grid_models[[i]])))
13 }
```

Example in Python

```
import h2o
from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
from h2o.grid.grid_search import H2OGridSearch
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/prostate/prostate.csv")
h2o_df['CAPSULE'] = h2o_df['CAPSULE'].asfactor()
alpha_opts = [0.0, 0.25, 0.5, 1.0]
hyper_parameters = {"alpha":alpha_opts}


grid = H2OGridSearch(H2OGeneralizedLinearEstimator(
    family="binomial"), hyper_params=hyper_parameters)
grid.train(y = "CAPSULE", x = ["AGE", "RACE", "PSA", "
    GLEASON"], training_frame = h2o_df)
for m in grid:
    print "Model ID: " + m.model_id + " auc: " , m.auc
        ()
    print m.summary()
    print "\n\n"
```

5.7.6 Grid Search Over Lambda

While automatic lambda search is the preferred method, a grid search over lambda values is also supported by passing in a vector of lambdas and disabling the lambda-search option. The behavior will be identical to lambda search, except H2O will use the specified list of lambdas instead (still capped at λ_{max}).

Example in R

```
library(h2o)
h2o.init()
path = system.file("extdata", "prostate.csv", package
    = "h2o")
h2o_df = h2o.importFile(path)
h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
```

```
lambda_opts = list(list(1), list(.5), list(.1), list(.01), list(.001), list(.0001), list(.00001), list(0))
hyper_parameters = list(lambda = lambda_opts)
grid <- h2o.grid("glm", hyper_params = hyper_parameters,
                 y = "CAPSULE", x = c("AGE", "RACE", "PSA", "GLEASON"), training_frame = h2o_df, family = "binomial")
grid_models <- lapply(grid@model_ids, function(model_id) { model = h2o.getModel(model_id) })
for (i in 1:length(grid_models)) {
    print(sprintf("regularization: %-50s  auc: %f", grid_models[[i]]@model$model_summary$regularization, h2o.auc(grid_models[[i]])))
}
```

Example in Python

```
import h2o
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
from h2o.grid.grid_search import H2OGridSearch
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.s3.amazonaws.com/smalldata/prostate/prostate.csv")
h2o_df['CAPSULE'] = h2o_df['CAPSULE'].asfactor()
lambda_opts = [1, 0.5, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0]
hyper_parameters = {"lambda":lambda_opts}


grid = H2OGridSearch(H2OGeneralizedLinearEstimator(family="binomial"), hyper_params=hyper_parameters)
grid.train(y = "CAPSULE", x = ["AGE", "RACE", "PSA", "GLEASON"], training_frame = h2o_df)
for m in grid:
    print "Model ID:", m.model_id, " auc:", m.auc()
    print m.summary()
    print "\n\n"
```

5.7.7 Offsets

offset_column is an optional column name or index referring to a column in the training frame. This column specifies a prior known component to be included in the linear predictor during training. Offsets are per-row "bias values" that are used during model training.

For Gaussian distributions, they can be seen as simple corrections to the response (y) column. Instead of learning to predict the response (y-row), the model learns to predict the (row) offset of the response column.

For other distributions, the offset corrections are applied in the linearized space before applying the inverse link function to get the actual response values.

5.7.8 Row Weights

weights_column is an optional column name or index referring to a column in the training frame. This column specifies on a per-row basis the weight of that row. If no weight column is specified, a default value of 1 is used for each row. Weights are per-row observation weights. This is typically the number of times a row is repeated, but non-integer values are supported as well. During training, rows with higher weights matter more, due to the larger loss function pre-factor.

5.7.9 Coefficient Constraints

Coefficient constraints allow you to set special conditions over the model coefficients. Currently supported constraints are upper and lower bounds and the proximal operator interface, as described in Proximal Algorithms by Boyd et. al.

The constraints are specified as a frame with following vecs (matched by name; all vecs can be sparse):

- names: (mandatory) coefficient names
- lower_bounds: (optional) coefficient lower bounds , must be less than or equal to upper_bounds
- upper_bounds: (optional) coefficient upper bounds , must be greater than or equal to lower_bounds
- beta_given: (optional) specifies the given solution in proximal operator interface

- `rho` (mandatory if `beta_given` is specified, otherwise ignored): specifies per-column ℓ_2 penalties on the distance from the given solution

5.7.10 Proximal Operators

The proximal operator interface allows you to run the GLM with a proximal penalty on a distance from a specified given solution. There are many potential uses: for example, it can be used as part of an ADMM consensus algorithm to obtain a unified solution over separate H2O clouds or in Bayesian regression approximation.

6 GLM Model Output

The following sections represent the output produced by logistic regression (i.e. binomial classification).

Example in R

```
library(h2o)
h2o.init()
path = system.file("extdata", "prostate.csv", package
    = "h2o")
h2o_df = h2o.importFile(path)
h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
rand_vec <- h2o.runif(h2o_df, seed = 1234)
train <- h2o_df[rand_vec <= 0.8,]
valid <- h2o_df[rand_vec > 0.8,]
binomial.fit = h2o.glm(y = "CAPSULE", x = c("AGE", "
    RACE", "PSA", "GLEASON"), training_frame = train,
    validation_frame = valid, family = "binomial")
print(binomial.fit)
```

Example in Python

```
import h2o
from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/prostate/prostate.csv")
```

```
h2o_df['CAPSULE'] = h2o_df['CAPSULE'].asfactor()
r = h2o_df[0].runif(seed=1234)
train = h2o_df[r <= 0.8]
valid = h2o_df[r > 0.8]
binomial_fit = H2OGeneralizedLinearEstimator(family =
    "binomial")
binomial_fit.train(y = "CAPSULE", x = ["AGE", "RACE",
    "PSA", "GLEASON"], training_frame = train,
    validation_frame=valid)
print binomial_fit
```

```
Model Details:
==============

H2OBinomialModel: glm
Model ID:  GLM_model_R_1439511782434_25
GLM Model:
    family  link
       regularization number_of_predictors_total
       number_of_active_predictors
       number_of_iterations training_frame
1 binomial logit Elastic Net (alpha = 0.5, lambda =
   4.674E-4 )                            4
                             5                    5
        subset_39

Coefficients:
      names coefficients standardized_coefficients
1 Intercept    -6.467393                 -0.414440
2       AGE    -0.021983                 -0.143745
3      RACE    -0.295770                 -0.093423
4       PSA     0.028551                  0.604644
5   GLEASON     1.156808                  1.298815

H2OBinomialMetrics: glm
** Reported on training data. **

MSE:  0.1735008
R^2:  0.2842015
LogLoss:  0.5151585
AUC:  0.806806
Gini:  0.6136121
Null Deviance:  403.9953
Residual Deviance:  307.0345
AIC:  317.0345

Confusion Matrix for F1-optimal threshold:
         0   1    Error     Rate
0      125  50 0.285714  =50/175
1       24  99 0.195122  =24/123
Totals 149 149 0.248322  =74/298

```

```
Maximum Metrics:
                        metric threshold    value idx
1                       max f1  0.301518 0.727941 147
2                       max f2  0.203412 0.809328 235
3                 max f0point5  0.549771 0.712831  91
4                 max accuracy  0.301518 0.751678 147
5                max precision  0.997990 1.000000   0
6           max absolute_MCC  0.301518 0.511199 147
7 max min_per_class_accuracy  0.415346 0.739837 134

H2OBinomialMetrics: glm
** Reported on validation data. **

MSE:  0.1981162
R^2:  0.1460683
LogLoss:  0.5831277
AUC:  0.7339744
Gini:  0.4679487
Null Deviance:  108.4545
Residual Deviance:  95.63294
AIC:  105.6329

Confusion Matrix for F1-optimal threshold:
        0  1    Error    Rate
0      35 17 0.326923  =17/52
1       8 22 0.266667   =8/30
Totals 43 39 0.304878  =25/82

Maximum Metrics:
                        metric threshold    value idx
1                       max f1  0.469237 0.637681  38
2                       max f2  0.203366 0.788043  63
3                 max f0point5  0.527267 0.616438  28
4                 max accuracy  0.593421 0.719512  18
5                max precision  0.949357 1.000000   0
6           max absolute_MCC  0.469237 0.391977  38
7 max min_per_class_accuracy  0.482906 0.692308  36
```

6.1 Coefficients and Normalized Coefficients

Coefficients are the predictor weights (i.e. the actual model used for prediction). Coefficients should be used to make predictions for new data points:

```
binomial.fit@model$coefficients
```

```
binomial_fit.coef()
```

```
  Intercept          AGE         RACE          PSA
     GLEASON
-6.46739299 -0.02198278 -0.29576986  0.02855057
   1.15680761
```

If the `standardize` option is enabled, H2O returns another set of coefficients: the standardized coefficients. These are the predictor weights of the standardized data and are included only for informational purposes (e.g. to compare relative variable importance).

In this case, the "normal" coefficients are obtained from the standardized coefficients by **reversing** the data standardization process (de-scaled, with the intercept adjusted by an added offset) so that they can be applied to data in its original form (i.e. no standardization prior to scoring). **Note:** These are **not** the same as coefficients of a model built on non-standardized data.

Standardized coefficients are useful for comparing the relative contribution of different predictors to the model:

```
binomial.fit@model$coefficients_table
```

```
binomial_fit.pprint_coef()
```

```
Coefficients:
      names coefficients standardized_coefficients
1 Intercept    -6.467393                 -0.414440
2       AGE    -0.021983                 -0.143745
3      RACE    -0.295770                 -0.093423
4       PSA     0.028551                  0.604644
5   GLEASON     1.156808                  1.298815
```

This view provides a sorted list of standardized coefficients in descending order for easy comparison:

```
binomial.fit@model$standardized_coefficient_magnitudes
```

```
sorted(binomial_fit.coef_norm().items(), key=lambda x:
    x[1], reverse=True)
```

```
Standardized Coefficient Magnitudes:
          names coefficients sign
GLEASON GLEASON     1.298815  POS
PSA         PSA     0.604644  POS
AGE         AGE     0.143745  NEG
RACE       RACE     0.093423  NEG
```

6.2 Model Statistics

Various model statistics are available:

`MSE` is the mean squared error: $MSE = \frac{1}{N}\sum_{i=1}^{N}(actual_i - prediction_i)^2$

`R^2` is the R squared: $R^2 = 1 - \frac{MSE}{\sigma_y^2}$

`LogLoss` is the log loss. $LogLoss = \frac{1}{N}\sum_{i}^{N}\sum_{j}^{C} y_i log(p_{i,j})$

`AUC` is available only for binomial models and is defined the area under ROC curve.

`Null deviance` Deviance (defined by selected family) computed for the null model.

`Residual deviance` Deviance of the built model

`AIC` is based on log-likelihood, which is summed up similarly to deviance

Retrieve these statistics using the following accessor functions:

Example in R

```
h2o.num_iterations(binomial.fit)
h2o.null_dof(binomial.fit, train = TRUE, valid = TRUE)
h2o.residual_dof(binomial.fit, train = TRUE, valid =
    TRUE)

h2o.mse(binomial.fit, train = TRUE, valid = TRUE)
h2o.r2(binomial.fit, train = TRUE, valid = TRUE)
```

```
h2o.logloss(binomial.fit, train = TRUE, valid = TRUE)
h2o.auc(binomial.fit, train = TRUE, valid = TRUE)
h2o.giniCoef(binomial.fit, train = TRUE, valid = TRUE)
h2o.null_deviance(binomial.fit, train = TRUE, valid =
    TRUE)
h2o.residual_deviance(binomial.fit, train = TRUE,
    valid = TRUE)
h2o.aic(binomial.fit, train = TRUE, valid = TRUE)
```

Example in Python

```
binomial_fit.summary()
binomial_fit._model_json["output"]["model_summary"].
    __getitem__('number_of_iterations')

binomial_fit.null_degrees_of_freedom(train=True, valid
    =True)
binomial_fit.residual_degrees_of_freedom(train=True,
    valid=True)

binomial_fit.mse(train=True, valid=True)
binomial_fit.r2(train=True, valid=True)
binomial_fit.logloss(train=True, valid=True)
binomial_fit.auc(train=True, valid=True)
binomial_fit.giniCoef(train=True, valid=True)
binomial_fit.null_deviance(train=True, valid=True)
binomial_fit.residual_deviance(train=True, valid=True)
binomial_fit.aic(train=True, valid=True)
```

6.3 Confusion Matrix

Fetch the confusion matrix directly using the following accessor function:

Example in R

```
h2o.confusionMatrix(binomial.fit, valid = FALSE)
h2o.confusionMatrix(binomial.fit, valid = TRUE)
```

Example in Python

```
binomial_fit.confusion_matrix(valid=False)
binomial_fit.confusion_matrix(valid=True)
```

6.4 Scoring History

The following output example represents a sample scoring history:

```
binomial.fit@model$scoring_history
```

Example in Python

```
binomial_fit.scoring_history
```

```
Scoring History:
            timestamp   duration iteration
               log_likelihood objective
1 2015-08-13 19:05:17  0.000 sec         0
   201.99764   0.67784
2 2015-08-13 19:05:17  0.002 sec         1
   158.46117   0.53216
3 2015-08-13 19:05:17  0.003 sec         2
   153.74404   0.51658
4 2015-08-13 19:05:17  0.004 sec         3
   153.51935   0.51590
5 2015-08-13 19:05:17  0.005 sec         4
   153.51723   0.51590
6 2015-08-13 19:05:17  0.006 sec         5
   153.51723   0.51590
```

7 Making Predictions

Once you have built a model, you can use it to make predictions using two different approaches: the in-H2O batch scoring approach and the real-time nano-fast POJO approach.

7.1 Batch In-H2O Predictions

Batch in-H2O predictions are made using a normal H2O cluster on a new H2OFrame. When you use `h2o.predict()`, the order of the rows in the results is the same as the order in which the data was loaded, even if some rows fail (for example, due to missing values or unseen factor levels). In addition to predictions, you can view metrics such as area under curve (AUC) if you include the response column in the new data. The following example represents a logistic regression model (i.e. binomial classification).

Example in R

```
library(h2o)
h2o.init()
path = system.file("extdata", "prostate.csv", package
    = "h2o")
h2o_df = h2o.importFile(path)
h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
rand_vec <- h2o.runif(h2o_df, seed = 1234)
train <- h2o_df[rand_vec <= 0.8,]
valid <- h2o_df[(rand_vec > 0.8) & (rand_vec <= 0.9),]
test <- h2o_df[rand_vec > 0.9,]
binomial.fit = h2o.glm(y = "CAPSULE", x = c("AGE", "
    RACE", "PSA", "GLEASON"), training_frame = train,
    validation_frame = valid, family = "binomial")

# Make and export predictions.
pred = h2o.predict(binomial.fit, test)
h2o.exportFile(pred, "/tmp/pred.csv", force = TRUE)
# Or you can export the predictions to hdfs:
#   h2o.exportFile(pred, "hdfs://namenode/path/to/file
    .csv")

# Calculate metrics.
perf = h2o.performance(binomial.fit, test)
print(perf)
```

Example in Python

```
h2o.init()
h2o_df = h2o.import_file("http://h2o-public-test-data.s3.amazonaws.com/smalldata/prostate/prostate.csv")
h2o_df['CAPSULE'] = h2o_df['CAPSULE'].asfactor()

rand_vec = h2o_df.runif(1234)

train = h2o_df[rand_vec <= 0.8]
valid = h2o_df[(rand_vec > 0.8) & (rand_vec <= 0.9)]
test = h2o_df[rand_vec > 0.9]
binomial_fit = H2OGeneralizedLinearEstimator(family = "binomial")
binomial_fit.train(y = "CAPSULE", x = ["AGE", "RACE", "PSA", "GLEASON"], training_frame = train, validation_frame = valid)

# Make and export predictions.
pred = binomial_fit.predict(test)
h2o.export_file(pred, "/tmp/pred.csv", force = True)
# Or you can export the predictions to hdfs:
#   h2o.exportFile(pred, "hdfs://namenode/path/to/file.csv")

# Calculate metrics.
binomial_fit.model_performance(test)
```

Here is an example of making predictions on new data:

```
# Remove the response column to simulate new data points arriving without the answer being known.
newdata = test
newdata$CAPSULE <- NULL
newpred = h2o.predict(binomial.fit, newdata)
head(newpred)
```

```
# Remove the response column to simulate new data points arriving without the answer being known.
newdata = test
newdata['CAPSULE'] = None
```

```
newpred = binomial_fit.predict(newdata)
newpred
```

```
  predict         p0        p1
1       1 0.1676892 0.8323108
2       0 0.4824181 0.5175819
3       1 0.2000061 0.7999939
4       0 0.9242169 0.0757831
5       0 0.5044669 0.4955331
6       0 0.7272743 0.2727257
```

The three columns in the prediction file are the predicted class, the probability that the prediction is class 0, and the probability that the prediction is class 1. The predicted class is chosen based on the maximum-F1 threshold.

You can change the threshold manually, for example to 0.3, and recalculate the predict column like this:

```
newpred$predict = newpred$p1 > 0.3
head(newpred)
```

```
#manually define threshold for predictions to 0.3
import pandas as pd
pred = binomial_fit.predict(h2o_df)
pred['predict'] = pred['p1']>0.3
```

```
  predict         p0        p1
1       1 0.1676892 0.8323108
2       1 0.4824181 0.5175819
3       1 0.2000061 0.7999939
4       0 0.9242169 0.0757831
5       1 0.5044669 0.4955331
6       0 0.7272743 0.2727257
```

7.2 Low-latency Predictions using POJOs

For nano-fast scoring, H2O GLM models can be directly rendered as a Plain Old Java Object (POJO). POJOs are very low-latency and can easily be embedded in any Java environment (a customer-facing web application, a Storm bolt, or a Spark Streaming pipeline, for example).

The POJO does nothing but pure math, and has no dependencies on any other software packages (not even H2O), so it is easy to implement.

Directions for using the POJO in detail are beyond the scope of this document, but the following example demonstrates how to generate and view a POJO. To access the POJO from the Flow Web UI, click the DOWNLOAD POJO button at the bottom of the cell containing the generated model.

For more information on how to use an H2O POJO, refer to the **POJO Quick Start Guide** at `https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/howto/POJO_QuickStart.md`.

Example in R

```
1 library(h2o)
2 h2o.init()
3 path = system.file("extdata", "prostate.csv", package
    = "h2o")
4 h2o_df = h2o.importFile(path)
5 h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
6 binomial.fit = h2o.glm(y = "CAPSULE", x = c("AGE", "
    RACE", "PSA", "GLEASON"), training_frame = h2o_df,
     family = "binomial")
7 h2o.download_pojo(binomial.fit)
```

Example in Python

```
1 import h2o
2 from h2o.estimators.glm import
    H2OGeneralizedLinearEstimator
3 h2o.init()
4 h2o_df = h2o.import_file("http://h2o-public-test-data.
    s3.amazonaws.com/smalldata/prostate/prostate.csv")
5 h2o_df['CAPSULE'] = h2o_df['CAPSULE'].asfactor()
6 binomial_fit = H2OGeneralizedLinearEstimator(family =
    "binomial")
7 binomial_fit.train(y = "CAPSULE", x = ["AGE", "RACE",
    "PSA", "GLEASON"], training_frame = h2o_df)
8 h2o.download_pojo(binomial_fit)
```

8 Best Practices

Here are a few rules of thumb to follow:

- Use symmetric nodes in your H2O cluster
- Impute data before running GLM
- The IRLSM solver works best on tall and skinny datasets
- If you have a wide dataset, use an ℓ_1 penalty to eliminate columns from the model
- If you have a wide dataset, use the L-BFGS solver
- When using lambda search, specify a value for `max_predictors` if the process takes too long. 90% of the time is spent on the larger models with the small lambdas, so specifying `max_predictors` can reduce this time
- Retain a small ℓ_2 penalty (i.e. ridge regression) for numerical stability (i.e. don't use `alpha` 1.0, use 0.95 instead)
- When using the IRLSM solver, larger nodes can help the ADMM (Cholesky decomposition) run faster

8.1 Verifying Model Results

To determine the accuracy of your model, use the following guidelines:

- Look for conspicuously different cross-validation results between folds:

  Example in R

```
1 library(h2o)
2 h2o.init()
3 h2o_df = h2o.importFile("http://s3.amazonaws.com/
    h2o-public-test-data/smalldata/airlines/
    allyears2k_headers.zip")
4 model = h2o.glm(y = "IsDepDelayed", x = c("Year",
    "Origin"), training_frame = h2o_df, family = "
    binomial", nfolds = 5)
5 print(paste("full model training auc:",
    model@model$training_metrics@metrics$AUC))
6 print(paste("full model cv auc:", model@model$
    cross_validation_metrics@metrics$AUC))
```

```
for (i in 1:5) {
    cv_model_name = model@model$cross_validation_
        models[[i]]$name
    cv_model = h2o.getModel(cv_model_name)
    print(paste("cv fold ", i, " training auc:",
        cv_model@model$training_metrics@metrics$
        AUC, " validation auc: ", cv_model@model$
        validation_metrics@metrics$AUC))
}
```

Example in Python

```
h2o_df = h2o.import_file("http://s3.amazonaws.com/
    h2o-public-test-data/smalldata/airlines/
    allyears2k_headers.zip")
model = H2OGeneralizedLinearEstimator(family = "
    binomial", nfolds = 5)
model.train(y = "IsDepDelayed", x = ["Year", "
    Origin"], training_frame = h2o_df)


print "full model training auc:", model.auc()
print "full model cv auc:", model.auc(xval=True)
for model_ in model.get_xval_models():
    print model_.model_id, " training auc:",
        model_.auc(), " validation auc: ", model_.
        auc(valid=True)
```

- Look for explained deviance ($1 - \frac{NullDev - ResDev}{NullDev}$)
  - Too close to 0: model doesn't predict well (underfitting)
  - Too close to 1: model predicts too well due to noisy data (overfitting)
- For logistic regression (i.e. binomial classification) models, look for AUC
  - Too close to 0.5: model doesn't predict well (underfitting)
  - Too close to 1: model predicts too well due to noisy data (overfitting)
- Look at the number of iterations or scoring history to see if GLM stops early for a specific lambda; performing all the iterations usually means the solution is not good. This is controlled by the `max_iterations` parameter.

- The fewer the NA values in your training data, the better; GLM will either skip or mean-impute rows with NA values. Always check degrees of freedom in the output model. Degrees of freedom is the number of observations used to train the model minus the size of the model. If this number is much smaller than expected, it is likely that too many rows have been excluded due to missing values.
  - If you have few columns with many NAs, you might accidentally be losing all your rows, so it's better to exclude them.
  - If you have many columns with small fraction of uniformly-distributed missing values, every row will likely have at least one missing value. In this case, impute the NAs (e.g. substituted with mean values) before modeling.

9 Implementation Details

The following sections discuss some of the implementation choices in H2O's GLM.

9.1 Categorical Variables

When applying linear models to datasets with categorical variables, the usual approach is to expand the categoricals into a set of binary vectors, with one vector per each categorical level (e.g. by calling `model.matrix` in R). H2O performs similar expansions automatically and no prior changes to the dataset are needed. Each categorical column is treated as a set of sparse binary vectors.

9.1.1 Largest Categorical Speed Optimization

Categoricals have special handling during GLM computation as well. When forming the gram matrix, we can take advantage of the fact that columns belonging to the same categorical never co-occur and the gram matrix region belonging to these columns will not have any non-zero elements outside of the diagonal.

This keeps it in sparse representation, taking only $O(N)$ elements instead of $O(N * N)$. Furthermore, the complexity of Choelsky decomposition of a matrix that starts with a diagonal region can be greatly reduced. H2O's GLM exploits these two facts to handle the largest categorical "for free". Therefore, when

analyzing the performance of GLM in the equation expressed above, we can subtract the size of the largest categoricals from the number of predictors.

$$N = \sum_{c \in C} (\|c.domain\|) - \arg\max_{c \in C} \|c.domain\| + \|Nums\|$$

9.2 Performance Characteristics

This section discusses the CPU and memory cost of each available non-experimental solver for running GLM.

9.2.1 IRLSM Solver

The implementation is based on iterative re-weighted least squares with an ADMM inner solver (as described in Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers by Boyd et. al) to deal with the ℓ_1 penalty. Every iteration of the algorithm consists of following steps:

1. Generate weighted least squares problem based on previous solution, i.e. vector of weights w and response z
2. Compute the weighted gram matrix $X^T W X$ and $X^T z$ vector
3. Decompose the gram matrix (Cholesky decomposition) and apply ADMM solver to solve the ℓ_1 penalized least squares problem

Steps 1 and 2 are performed distributively. Step 3 is computed in parallel on a single node. This method characterizes the computational complexity and scalability of a dataset with M observations and N columns (predictors) on a cluster with n nodes with p CPUs each.

	CPU	Memory
Gram matrix ($X^T X$) (distributed)	$O(\frac{MN^2}{pn})$	O(training data) + O(gram matrix) $O(MN) + O(N^2 pn)$
ADMM + Cholesky decomposition (single node)	$O(\frac{N^3}{p})$	$O(N^2)$

M	Number of rows in the training data
N	Number of predictors in the training data
p	Number of CPUs per node
n	Number of nodes in the cluster

If $M >> N$, the algorithm scales linearly both in the number of nodes and the number of CPUs per node. However, the algorithm is limited in the number of predictors it can handle, since the size of the Gram matrix grows quadratically, due to a memory and network throughput issue with the number of predictors.

Its decomposition cost grows as the cube of the number of predictors increases, which is a computational cost issue. In many cases, H2O can work around these limitations due to its handling of categoricals and by employing strong rules to filter out inactive predictors.

9.2.2 L-BFGS solver

In each iteration, L-BFGS computes a gradient at the current vector of coefficients and then computes an updated vector of coefficients in an approximated Newton-method step.

The cost of the coefficient update is $k * N$, where N is number of predictors and k is a constant, the cost of gradient computation is $\frac{M*N}{pn}$ where M is number of observations in the dataset and pn is the number of CPU cores in the cluster. Since k is a small constant, the runtime of L-BFGS is dominated by the gradient computation, which is fully parallelized, scaling L-BFGS almost linearly.

9.3 FAQ

- **What if the training data contains NA values?**

  The rows with missing response are ignored during model training and validation.

- **What if the testing data contains NA values?**

  If the missing value handling is set to skip and you are generating predictions, skipped rows will have NA (missing) prediction.

- **What if, while making predictions on testing data, a predictor column is categorical and the predictor is a level not observed during training?**

  The value is zero for all predictors associated with that categorical variable.

- **What if, while making predictions on testing data, the response column is categorical and the response is a level not observed during training?**

  H2O supports binomial models only; any extra levels in the test response will generate an error.

10 Appendix: Parameters

- `x`: A vector containing the names of the predictors to use while building the GLM model. No default.
- `y`: A character string or index that represents the response variable in the model. No default.
- `training_frame`: An `H2OFrame` object containing the variables in the model.
- `model_id`: (Optional) The unique ID assigned to the generated model. If not specified, an ID is generated automatically.
- `validation_frame`: An `H2OParsedData` object containing the validation dataset used to construct confusion matrix. If blank, the training data is used by default.
- `max_iterations`: A non-negative integer specifying the maximum number of iterations.
- `objective_epsilon`: Specify a threshold for convergence. If the objective value is less than this threshold, the model is converged.
- `beta_epsilon`: A non-negative number specifying the magnitude of the maximum difference between the coefficient estimates from successive iterations. Defines the convergence criterion.
- `gradient_epsilon`: (For L-BFGS only) Specify a threshold for convergence. If the objective value (using the L-infinity norm) is less than this threshold, the model is converged.
- `solver`: A character string specifying the solver used: either `IRLSM`, which supports more features, or `L_BFGS`, which scales better for datasets with many columns.
- `standardize`: A logical value that indicates whether the numeric predictors should be standardized to have a mean of 0 and a variance of 1 prior to model training.

- `family`: A description of the error distribution and corresponding link function to be used in the model. The following options are supported: `gaussian`, `binomial`, `gamma`, `multinomial`, `poisson`, `tweedie`, or `quasibinomial`. When a model is specified as Tweedie, users must also specify the appropriate Tweedie power. No default.
- `link`: The link function relates the linear predictor to the distribution function. The default is the canonical link for the specified family. The full list of supported links:
  - **gaussian**: identity, log, inverse
  - **binomial**: logit
  - **quasibinomial**: logit
  - **multinomial**: multinomial (family_default)
  - **poisson**: log, identity
  - **gamma**: inverse, log, identity
  - **tweedie**: tweedie
- `tweedie_variance_power`: A numeric specifying the power for the variance function when `family = "tweedie"`. Default is 0.
- `tweedie_link_power`: A numeric specifying the power for the link function when `family = "tweedie"`. Default is 1.
- `alpha`: The elastic-net mixing parameter, which must be in $[0, 1]$. The penalty is defined to be $P(\alpha, \beta) = (1 - \alpha)/2||\beta||_2^2 + \alpha||\beta||_1 = \sum_j[(1 - \alpha)/2\beta_j^2 + \alpha|\beta_j|]$ so `alpha=1` is the Lasso penalty, while `alpha=0` is the ridge penalty. Default is 0.5.
- `prior`: (Optional) A numeric specifying the prior probability of class 1 in the response when `family = "binomial"`. The default value is the observation frequency of class 1. Must be from (0,1) exclusive range or NULL (no prior).
- `lambda`: A non-negative value representing the shrinkage parameter, which multiplies $P(\alpha, \beta)$ in the objective. The larger lambda is, the more the coefficients are shrunk toward zero (and each other). When the value is 0, regularization is disabled and ordinary generalized linear models are fit. The default is 1e-05.
- `lambda_search`: A logical value indicating whether to conduct a search over the space of lambda values, starting from the max lambda, given lambda will be interpreted as the min. lambda. Default is false.

- nlambdas: The number of lambda values when lambda_search = TRUE. Default is -1.

- lambda_min_ratio: Smallest value for lambda as a fraction of lambda.max, the entry value, which is the smallest value for which all coefficients in the model are zero. If the number of observations is greater than the number of variables then lambda_min_ratio = 0.0001; if the number of observations is less than the number of variables then lambda_min_ratio = 0.01. Default is -1.0.

- nfolds: Number of folds for cross-validation. If nfolds >=2, then validation_frame must remain blank. Default is 0.

- fold_column: (Optional) Column with cross-validation fold index assignment per observation.

- fold_assignment: Cross-validation fold assignment scheme, if fold_column is not specified. The following options are supported: AUTO, Random, or Modulo.

- keep_cross_validation_predictions: Specify whether to keep the predictions of the cross-validation models.

- beta_constraints: A data frame or H2OParsedData object with the columns ["names", "lower_bounds", "upper_bounds", "beta_given"], where each row corresponds to a predictor in the GLM. "names" contains the predictor names, "lower_bounds"/ "upper_bounds" are the lower and upper bounds (respectively) of the beta, and "beta_given" is a user-specified starting value.

- offset_column: Specify the offset column. Note: Offsets are per-row bias values that are used during model training. For Gaussian distributions, they can be seen as simple corrections to the response (y) column. Instead of learning to predict the response (y-row), the model learns to predict the (row) offset of the response column. For other distributions, the offset corrections are applied in the linearized space before applying the inverse link function to get the actual response values.

- weights_column: Specify the weights column. Note: Weights are per-row observation weights. This is typically the number of times a row is repeated, but non-integer values are supported as well. During training, rows with higher weights matter more, due to the larger loss function pre-factor.

- `intercept`: Logical; includes a constant term (intercept) in the model. If there are factor columns in your model, then the intercept must be included.
- `max_runtime_secs`: Maximum allowed runtime in seconds for model training. Use 0 to disable.
- `missing_values_handling`: Handling of missing values. Either `Skip` or `MeanImputation` (default).
- `seed`: Specify the random number generator (RNG) seed for algorithm components dependent on randomization. The seed is consistent for each H2O instance so that you can create models with the same starting conditions in alternative configurations.
- `max_active_predictors`: Specify the maximum number of active predictors during computation. This value is used as a stopping criterium to prevent expensive model building with many predictors.
- `compute_p_values`: Request GLM to compute p-values. This is only applicable with no penalty (lambda = 0 and no beta constraints). Setting `remove_collinear_columns` is recommended. H2O will return an error if p-values are requested when there are collinear columns and the `remove_collinear_columns` flag is not enabled.
- `non_negative`: Forces coefficients to have non-negative values.
- `remove_collinear_columns`: Specify whether to automatically remove collinear columns during model building. When enabled, collinear columns will be dropped from the model and will have a 0 coefficient in the returned model. This can only be set if there is no regularization (lambda=0).
- `interactions`: Optionally specify a list of predictor column indices to interact. All pairwise combinations will be computed for this list.

11 References

Jerome Friedman, Trevor Hastie, and Rob Tibshirani. **Regularization Paths for Generalized Linear Models via Coordinate Descent**. *Journal of Statistical Software*, 33(1), 2009. URL http://core.ac.uk/download/pdf/6287975.pdf

Jacob Bien, Jerome Friedman, Trevor Hastie, Noah Simon, Jonathan Taylor, Rob Tibshirani, and Ryan J. Tibshirani. **Strong Rules for Discarding Predictors in Lasso-type Problems**. *Journal of the Royal Statistical Society. Series B*, 74(1), 2012. URL http://statweb.stanford.edu/~tibs/ftp/strong.pdf

Hui Zou and Trevor Hastie. **Regularization and variable selection via the Elastic Net**. *Journal of the Royal Statistical Society. Series B*, 67:301–320, 2005

Rob Tibshirani. **Regression Shrinkage and Selection via the Lasso**. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. URL http://statweb.stanford.edu/~tibs/lasso/lasso.pdf

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonahtan Eckstein. **Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers**. *Foundations and Trends in Machine Learning*, 3(1):1–122, 1996. URL https://web.stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf

Neal Parikh and Stephen Boyd. **Proximal Algorithms**. *Foundations and Trends in Optimization*, 1(3):123–231, 2014. URL http://web.stanford.edu/~boyd/papers/pdf/prox_algs.pdf

H2O.ai Team. **H2O website**, 2016. URL http://h2o.ai

H2O.ai Team. **H2O documentation**, 2016. URL http://docs.h2o.ai

H2O.ai Team. **H2O GitHub Repository**, 2016. URL https://github.com/h2oai

H2O.ai Team. **H2O Datasets**, 2016. URL http://data.h2o.ai

H2O.ai Team. **H2O JIRA**, 2016. URL https://jira.h2o.ai

H2O.ai Team. **H2Ostream**, 2016. URL https://groups.google.com/d/forum/h2ostream

H2O.ai Team. **H2O R Package Documentation**, 2016. URL http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Rdoc.html

12 Authors

Tomas Nykodym

Tomas is our resident Software Engineer. He received his Masters degree from the Czech Technical University. Tomas has worked at IBM-research and Agent-Technology Group. Tomas also created a sandbox with simulated user activity for safe execution of malware with advanced behavior extraction algorithms to extract behavior of malware injected into other processes.

Tom Kraljevic

Tom is VP of Engineering of Customer and Pre-Sales Engineering at H2O and key to the magic of engineering and customer happiness. Tom has an MS degree in Electrical Engineering from the University of Illinois (at Urbana-Champaign), and a BSE degree in Computer Engineering from the University of Michigan (at Ann Arbor).

Nadine Hussami

Nadine is a math hacker intern at H2O. She is a PhD student at Stanford University. Her main research interests include statistical learning and convex optimization. Before Stanford, she worked as an algorithm development engineer at Bina Technologies. She holds a Masters degree from Stanford in Electrical Engineering.

Ariel Rao

Ariel is a data and math hacker intern at H2O. She is currently an undergraduate student at Carnegie Mellon University pursing degrees in Logic and Computation and Computer Science. Her interests include optimizing automated computation and formal verification.

Amy Wang

Amy is the Sales Engineering Lead at H2O. She graduated from Hunter College in NYC with a Masters in Applied Mathematics and Statistics with a heavy concentration on numerical analysis and financial mathematics. Her interest in applicable math eventually lead her to big data and finding the appropriate mediums for data analysis.

Jessica Lanford

Jessica is a word hacker and seasoned technical communicator at H2O.ai. Having worked for some of the top companies in technology including Dell, AT&T, and Lam Research, she is an expert at translating complex ideas to digestible articles.