Paper SAS-2017

Stacked Ensemble Models for Improved Prediction Accuracy

Funda Güneş, Russ Wolfinger, and Pei-Yi Tan

SAS Institute Inc.

ABSTRACT

Ensemble modeling is now a well-established means for improving prediction accuracy; it enables you to average out noise from diverse models and thereby enhance the generalizable signal. Basic stacked ensemble techniques combine predictions from multiple machine learning algorithms and use these predictions as inputs to second-level learning models. This paper shows how you can generate a diverse set of models by various methods such as forest, gradient boosted decision trees, factorization machines, and logistic regression and then combine them with stacked-ensemble techniques such as hill climbing, gradient boosting, and nonnegative least squares in SAS® Visual Data Mining and Machine Learning. The application of these techniques to real-world big data problems demonstrates how using stacked ensembles produces greater prediction accuracy and robustness than do individual models. The approach is powerful and compelling enough to alter your initial data mining mindset from finding the single best model to finding a collection of really good complementary models. It does involve additional cost due both to training a large number of models and the proper use of cross validation to avoid overfitting. This paper shows how to efficiently handle this computational expense in a modern SAS® environment and how to manage an ensemble workflow by using parallel computation in a distributed framework.

INTRODUCTION

Ensemble methods are commonly used to boost predictive accuracy by combining the predictions of multiple machine learning models. Model stacking is an efficient ensemble method in which the predictions that are generated by using different learning algorithms are used as inputs in a second-level learning algorithm. This second-level algorithm is trained to optimally combine the model predictions to form a final set of predictions (Sill et al. 2009).

In the last decade, model stacking has been successfully used on a wide variety of predictive modeling problems to boost the models' prediction accuracy beyond the level obtained by any of the individual models. This is sometimes referred to as a "wisdom of crowds" approach, pulling from the age-old philosophy of Aristotle. Ensemble modeling and model stacking are especially popular in data science competitions, in which a sponsor posts training and test data and issues a global challenge to produce the best model for a specified performance criterion. The winning model is almost always an ensemble model. Often individual teams develop their own ensemble model in the early stages of the competition and then join forces in the later stages. One such popular site is Kaggle, and you are encouraged to explore numerous winning solutions that are posted in the discussion forums there to get a flavor of the state of the art.

The diversity of the models in a library plays a key role in building a powerful ensemble model. Dietterich (2000) emphasizes the importance of diversity by stating, "A necessary and sufficient condition for an ensemble model to be more accurate than any of its individual members is if the classifiers are accurate and diverse." By combining information from diverse modeling approaches, ensemble models gain more accuracy and robustness than a fine-tuned single model can gain. There are many parallels with successful human teams in business, science, politics, and sports, in which each team member makes a significant contribution and individual weaknesses and biases are offset by the strengths of other members.

Overfitting is an omnipresent concern in ensemble modeling because a model library includes so many models that predict the same target. As the number of models in a model library increases, the chances of building overfitting ensemble models increases greatly. A related problem is leakage, in which

information from the target inadvertently and sometimes surreptitiously works its way into the model-checking mechanism and causes an overly optimistic assessment of generalization performance. The most efficient techniques that practitioners commonly use to minimize overfitting and leakage include cross validation, regularization, and bagging. This paper covers applications of these techniques for building ensemble models that can generalize well to new data.

This paper first provides an introduction to SAS Visual Data Mining and Machine Learning in SAS® Viya™, which is a new single, integrated, in-memory environment. The section following that discusses how to generate a diverse library of machine learning models for stacking while avoiding overfitting and leakage, and then shows an approach to building a diverse model library for a binary classification problem. A subsequent section shows how to perform model stacking by using regularized regression models, including nonnegative least squares regression. Another section demonstrates stacking with the scalable gradient boosting algorithm and focuses on an automatic tuning implementation that is based on efficient distributed and parallel paradigms for training and tuning models in the SAS Viya platform. The penultimate section shows how to build powerful ensemble models with the hill climbing technique. The last section compares the stacked ensemble models that are built by each approach to a naïve ensemble model and the single best model, and also provides a brief summary.

OVERVIEW OF THE SAS VIYA ENVIRONMENT

The SAS programs used in this paper are built in the new SAS Viya environment. SAS Viya uses SAS® Cloud Analytic Services (CAS) to perform tasks and enables you to build various model scenarios in a consistent environment, resulting in improved productivity, stability, and maintainability. SAS Viya represents a major rearchitecture of core data processing and analytical components in SAS software to enable computations across a large distributed grid in which it is typically more efficient to move algorithmic code rather than to move data.

The smallest unit of work for the CAS server is a CAS action. CAS actions can load data, transform data, compute statistics, perform analytics, and create output. Each action is configured by specifying a set of input parameters. Running a CAS action in the CAS server processes the action's parameters and the data to create an action result.

In SAS Viya, you can run CAS actions via a variety of interfaces, including the following:

- SAS session, which uses the CAS procedure. PROC CAS uses the CAS language (CASL) for specifying CAS actions and their input parameters. The CAS language also supports normal program logic such as conditional and looping statements and user-written functions.
- Python or Lua, which use the SAS Scripting Wrapper for Analytics Transfer (SWAT) libraries
- Java, which uses the CAS Client class
- Representational state transfer (REST), which uses the CAS REST APIs

CAS actions are organized into action sets, where each action set defines an application programming interface (API). SAS Viya currently provides the following action sets:

- Data mining and machine learning action sets support gradient boosted trees, neural networks, factorization machines, support vector machines, graph and network analysis, text mining, and more.
- Statistics action sets compute summary statistics and perform clustering, regression, sampling, principal component analysis, and more.
- Analytics action sets provide additional numeric and text analytics.
- System action sets run SAS code via the DATA step or DS2, manage CAS libraries and tables, manage CAS servers and sessions, and more.

SAS Viya also provides CAS-powered procedures, which enable you to have the familiar experience of coding traditional SAS procedures. Behind each statement in these procedures is one or more CAS

actions that run across multiple machines. The SAS Viya platform enables you to program with both CAS actions and procedures, providing you with maximum flexibility to build an optimal ensemble.

SAS Visual Data Mining and Machine Learning integrates CAS actions and CAS-powered procedures and surfaces in-memory machine-learning techniques such as gradient boosting, factorization machines, neural networks, and much more through its interactive visual interface, SAS® Studio tasks, procedures, and a Python client. This product bundle is an industry-leading platform for analyzing complex data, building predictive models, and conducting advanced statistical operations (Wexler, Haller, and Myneni 2017).

For more information about SAS Viya and SAS Visual Data Mining and Machine Learning, see the section "Recommended Reading." For specific code examples from this paper, refer to the Github repository referenced in that section.

BUILDING A STRONG LIBRARY OF DIVERSE MODELS

You can generate a diverse set of models by using many different machine learning algorithms at various hyperparameter settings. Forest and gradient bosting methods are themselves based on the idea of combining diverse decision tree models. The forest method generates diverse models by training decision trees on a number of bootstrap samples of the training set, whereas the gradient boosting method generates a diverse set of models by fitting models to sequentially adjusted residuals, a form of stochastic gradient descent. In a broad sense, even multiple regression models can be considered to be an ensemble of single regression models, with weights determined by least squares. Whereas the traditional wisdom in the literature is to combine so-called "weak" learners, the modern approach is to create an ensemble of a well-chosen collection of strong yet diverse models.

In addition to using many different modeling algorithms, the diversity in a model library can be further enhanced by randomly subsetting the rows (observations) and/or columns (features) in the training set. Subsetting rows can be done with replacement (bootstrap) or without replacement (for example, *k*-fold cross validation). The word "bagging" is often used loosely to describe such subsetting; it can also be used to describe subsetting of columns. Columns can be subsetted randomly or in a more principled fashion that is based on some computed measure of importance. The variety of choices for subsetting columns opens the door to the large and difficult problem of feature selection.

Each new big data set tends to bring its own challenges and intricacies, and no single fixed machine learning algorithm is known to dominate. Furthermore, each of the main classes of algorithms has a set of hyperparameters that must be specified, leading to an effectively infinite set of possible models you can fit. In order to navigate through this model space and achieve near optimal performance for a machine learning task, a basic brute-force strategy is to first build a reasonably large collection of model fits across a well-designed grid of settings and then compare, reduce, and combine them in some intelligent fashion. A modern distributed computing framework such as SAS Viya makes this strategy quite feasible.

AVOIDING LEAKAGE WHILE STACKING

A naïve ensembling approach is to directly take the predictions of the test data from a set of models that are fit on the full training set and use them as inputs to a second-level model, say a simple regression model. This approach is almost guaranteed to overfit the data because the target responses have been used twice, a form of data leakage. The resulting model almost always generalizes poorly for a new data set that has previously unseen targets. The following subsections describe the most common techniques for combatting leakage and selecting ensembles that will perform well on future data.

SINGLE HOLDOUT VALIDATION SET

The classic way to avoid overfitting is to set aside a fraction of the training data and treat its target labels as unseen until final evaluation of a model fitting process. This approach has been the main one available in SAS Enterprise Miner from its inception, and it remains a simple and reliable way to assess model accuracy. It can be the most efficient way to compare models. It also is the way most data science

competitions are structured for data sets that have a large number of rows.

For stacked ensembling, this approach also provides a good way to assess ensembles that are made on the dedicated training data. However, it provides no direct help in constructing those ensembles, nor does it provide any measure of variability in the model performance metric because you obtain only a single number. The latter concern can be addressed by scoring a set of bootstrap or other well-chosen random samples of the single holdout set.

K-FOLD CROSS VALIDATION AND OUT-OF-FOLD PREDICTIONS

The main idea of cross validation is to repeat the single holdout concept across different folds of the data—that is, to sequentially train a model on one part of the data and then observe the behavior of this trained model on the other held-out part, for which you know the ground truth. Doing so enables you to simulate performance on previously unseen targets and aims to decrease the bias of the learners with respect to the training data.

Assuming that each observation has equal weight, it makes sense to hold out each with equal frequency. The original jackknife (leave-one-out cross validation) method in regression holds out one observation at a time, but this method tends to be computationally infeasible for more complex algorithms and large data sets. A better approach is to hold out a significant fraction of the data (typically 10 or 20%) and divide the training data into *k* folds, where *k* is 5 or 10. The following simple steps are used to obtain five-fold cross validated predictions:

1. Divide the training data into five disjoint folds of as nearly equal size as possible, and possibly also stratify by target frequencies or means.
2. Hold out each fold one at a time.
3. Train the model on the remaining data.
4. Assess the trained model by using the holdout set.

Fitting and scoring for all *k* versions of the training and holdout sets provides holdout (cross validated) predictions for each of the samples in your original training data. These are known as out-of-fold (OOF) predictions. The sum of squared errors between the OOF predictions and true target values yields the cross validation error of a model, and is typically a good measure of generalizability. Furthermore, the OOF predictions are usually safely used as inputs for second-level stacked ensembling.

You might be able to further increase the robustness of your OOF predictions by repeating the entire *k*-fold exercise, recomputing OOFs with different random folds, and averaging the results. However, you must be careful to avoid possible subtle leakage if too many repetitions are done. Determining the best number of repetitions is not trivial. You can determine the best number by doing nested *k*-fold cross validation, in which you perform two-levels of *k*-fold cross validation (one within the other) and assess performance at the outer level. In this nested framework, the idea is to evaluate a small grid of repetition numbers, determine which one performs best, and then use this number for subsequent regular *k*-fold evaluations. You can also use this approach to help choose *k* if you suspect that the common values of 5 or 10 are suboptimal for your data.

Cross validation can be used both for tuning hyperparameters and for evaluating model performance. When you use the same data both for tuning and for estimating the generalization error with *k*-fold cross validation, you might have information leakage and the resulting model might overfit the data. To deal with this overfitting problem, you can use nested *k*-fold cross validation—you use the inner loop for parameter tuning, and you use the outer loop to estimate the generalization error (Cawley and Talbot 2010).

BAGGING AND OUT-OF-BAG PREDICTIONS

A technique similar in spirit to *k*-fold cross-validation is classical bagging, in which numerous bootstrap samples (with replacement) are constructed and the out-of-bag (OOB) predictions are used to assess model performance. One potential downside to this approach is the uneven number of times each

observation is held out and the potential for some missing values. However, this downside is usually inconsequential if you perform an appropriate number of bootstrap repetitions (for example, 100). This type of operation is very suitable for parallel processing, where with the right framework generating 100 bootstrap samples will not take much more clock time than 10 seconds.

AN APPROACH TO BUILDING A STRONG, DIVERSE MODEL LIBRARY

EXAMPLE: ADULT SALARY DATA SET

This section describes how to build a strong and diverse model library by using the Adult data set from the UCI Machine Learning Repository (Lichman 2013). This data set has 32,561 training samples and16,281 test samples; it includes 13 input variables, which are a mix of nominal and interval variables that include education, race, marital status, capital gain, and capital loss. The target is a binary variable that takes a value of 1 if a person makes less than 50,000 a year and value of 0 otherwise. The training and test set are available in a GitHub repository, for which a link is provided in the section "Recommended Reading."

Treating Nominal Variables

The data set includes six nominal variables that have various levels. The cardinality of the categorical variables is reduced by collapsing the rare categories and making sure that each distinct level has at least 2% of the samples. For example, the cardinality of the work class variable is reduced from 8 to 7, and the cardinality of the occupation variable is reduced from 14 to 12.

The nominal variable *education* is dropped from the analysis, because the corresponding interval variable (*education_num*) already exists. All the remaining nominal variables are converted to numerical variables by using likelihood encoding as described in the next section.

Likelihood Encoding and Feature Engineering

Likelihood encoding involves judiciously using the target variable to create numeric versions of categorical features. The most common way of doing this is to replace each level of the categorical variable with the mean of the target over all observations that have that level. Doing this carries a danger of information leakage that might result in significant overfitting. The best way to combat the danger of leakage is to perform the encoding separately for each distinct version of the training data during cross validation. For example, while doing five-fold cross validation, you compute the likelihood-encoded categorical variable anew for each of the five training sets and use these values in the corresponding holdout sets. A drawback of this approach is the extra calculations and bookkeeping that are required.

If the cardinality of a categorical variable is small relative to the number of observations and if the binary target is not rare, it can be acceptable to do the likelihood encoding once up front and run the risk of a small amount of leakage. For the sake of illustration and convenience, that approach is taken here with the Adult data set, because the maximum cardinality of the nominal variables is 12.

Likelihood encoding has direct ties to classical statistical methods such as one-way ANOVA, and it can be viewed as stacking the simple predictions from such models. More sophisticated versions involve shrinking the encoded means toward an overall mean, which can be particularly effective when the class sizes are imbalanced. This approach is well-known to improve mean square prediction error and is popularly known as L2 regularization in machine learning communities and as ridge regression or best linear unbiased prediction (BLUP) in statistical communities. Alternatively, you can use an L1 (LASSO) norm and shrink toward the median. Note also that likelihood encoding effectively performs the same operation that tree-based methods perform at their first step—that is, sorting categories by their target likelihood in order to find the best way to split them into two groups.

Stacking and Building the Model Library

As an illustrative small example, you can use the following three-level stacked ensemble approach along with four different machine learning algorithms (gradient boosting, forest, factorization machines, and logistic regression):

Level 1: Fit initial models and find good hyperparameters using cross validation and automatic tuning (also called autotuning).

Level 2: Create 100 bootstrap samples of the training set, and subsequently divide each of these samples into five folds. For each individual training set, train the four models (by using five-fold cross validation) and create 100 sets of five-fold OOF predictions. This approach effectively creates 400 total OOF predictions with approximately 1/3 of the values missing because of the properties of bootstrap (with replacement) sampling.

Level 3: Average together the nonmissing OOF predictions for each learning algorithm, creating four total average OOF predictions (one for each learning algorithm). Use LASSO, nonnegative least squares, gradient boosting, and hill climbing on these four features to obtain the final predictions.

As you move through the levels, you also create features on the final testing data. It is usually wise to keep training and testing features close to each other while coding. Otherwise you increase the risk of making a mistake at testing time because of an oversight in indexing or scoring. This practice also helps you keep your final goal in mind and ensure that everything you are doing is applicable to unlabeled testing rows.

Results for Level 1

Level 1 creates an initial small diverse library of models by using gradient boosting, forest, factorization machines, and logistic regression on the SAS Viya platform, which trains and tunes models quickly via in-memory processing by taking advantage of both multithreading and distributed computing. These algorithms include a fair number of hyperparameters that must be specified, and a manual tuning process can be difficult. Instead, you can use the efficient random search capability in the AUTOTUNE statement available in the GRADBOOST (scalable gradient boosting), FOREST, and the FACTMAC (factorization machines) procedures. By using autotuning, you can rapidly reduce the model error that is produced by default settings of these hyperparameters. This automated search provides an efficient search path through the hyperparameter space by taking advantage of parallel computing in the SAS Viya platform. The AUTOTUNE statement is also available in the NNET (neural network), TREESPLIT (decision tree), and SVMACHINE (support vector machine) procedures of SAS Viya Data Mining and Machine Learning. You can see an example of how autotuning is used in the section “Stacking with the Scalable Gradient Boosting Algorithm.” You must be wary of overfitting and leakage while doing this tuning. For more information about automated search, see Koch et al. (2017).

Results for Level 2

After finding good set of hyperparameter values for each of the four modeling algorithms, Level 2 generates 100 bootstrap replications (sampling with replacement) of the training data. Each training set is then divided into five disjoint folds, which produces five versions of new training sets (each version omits one fold) for each of the bootstrap samples. Notice that this setup produces 500 (100 x 5) versions of training sets. Forest, gradient boosting, factorization machine, and logistic regression models are trained on each of these training sets and the left-out folds are scored. In total, 2,000 (500 x 4) models are trained and scored. For each bootstrap sample, the five sets of OOF predictions are combined, which produces 400 columns of five-fold OOF predictions (100 gradient boosting, 100 forest, 100 logistic models, and 100 factorization machines).

Because bootstrap sampling uses sampling with replacement, it results in some missing predictions in addition to multiple predictions for the same IDs. This example adopts the following approach to deal with these issues and arrive at one prediction for each ID:

- If an ID is selected more than once, the average prediction is used for each ID.
- After making sure that each ID is selected at least once in the 100 bootstrap samples of each modeling algorithm, mean OOF predictions are obtained by averaging over 100 bootstrap OOF predictions. This simple averaging provided a significant reduction in the five-fold training ASE. For example, for the gradient boosting model, the five-fold training ASE of the best model (out of 100 models) was 0.09351. When the OOF predictions of 100 gradient boosting models are averaged, this value reduced to 0.09236.

This approach produces four columns of OOF predictions (one for each of the four algorithms). These four averaged models form the model library to be used in Level-3 stacking.

For scoring on test data, the predictions from the 500 models, which are generated by the same learning algorithm, are simply averaged.

Figure 1 shows the five-fold cross validation and test average squared errors (ASEs, also often called mean squared error, or MSE) of the four average models that form the model library to be used in Level-3 stacking. The best performing single modeling method is the average gradient boosting model, which has a five-fold cross validation ASE of 0.09236. It is best by a fairly significant margin according to the ASE performance metric.

Level-2 Models	Training ASE (Five-Fold CV ASE)	Testing ASE
Average gradient boosting	0.09236	0.09273
Average forest	0.09662	0.09665
Average logistic regression	0.10470	0.10370
Average factorization machines	0.11160	0.10930

Figure 1. Five-Fold Cross Validation and Test ASEs of Models in the Model Library

Results for Level 3

With average OOF predictions in hand from Level 2, you are ready to build final ensembles and assess the resulting models by using the test set predictions. The OOF predictions are stored in the SAS data set train_mean_oofs, which includes four columns of OOF predictions for the four average models, an ID variable, and the target variable. The corresponding test set is test_mean_preds which includes the same columns. The rest of the analyses in this paper use these two data sets, which are also available in the GitHub repository.

Start a CAS Session and Load Data into CAS

The following SAS code starts a CAS session and loads data into in the CAS in-memory distributed computing engine in the SAS Viya environment:

```
/* Start a CAS session named mySession */
cas mySession;

/* Define a CAS engine libref for CAS in-memory data tables  */
/* Define a SAS libref for the directory that includes the data */
libname cas sasioca;
libname data "/folders/myfolders/";

/* Load data into CAS using SAS DATA steps */
data cas.train_oofs;
   set data.train_mean_oofs;
run;
data cas.test_preds;
   set data.test_mean_preds;
run;
```

REGRESSION STACKING

Let Y represent the target, X represent the space of inputs, and $g_1, \ldots, g_L$ denote the learned predictions from L machine learning algorithms (for example, a set of out-of-fold predictions). For an interval target, a linear ensemble model builds a prediction function,

$$b(g) = w_1 * g_1 + \cdots + w_L * g_L$$

where w_i are the model weights. A simple way to specify these weights is to set them all equal to $1/L$ (as done in Level-2) so that each model contributes equally to the final ensemble. You can alternatively assign higher weight to models you think will perform better. For the Adult example, the gradient boosted tree OOF predictor is a natural candidate to weight higher because of its best single model performance.

Although assigning weights by hand can often be reasonable, you can typically improve final ensemble performance by using a learning algorithm to estimate them. Because of its computational efficiency and model interpretability, linear regression is a commonly used method for final model stacking. In a regression model that has an interval target, the model weights (w_i) are found by solving the following least squares problem:

$$min \sum_{i=1}^{N} (y_i - (w_1 * g_{1i} + \cdots + w_L * g_{Li}))^2$$

REGULARIZATION

Using cross validated predictions partially helps to deal with the overfitting problem. An attending difficulty with using OOF or OOB predictions as inputs is that they tend to be highly correlated with each other, creating the well-known collinearity problem for regression fitting. Arguably the best way to deal with this problem is to use some form of regularization for the model weights when training the highest-level model. Regularization methods place one or more penalties on the objective function, based on the size of the model weights. If these penalty parameters are selected correctly, the total prediction error of the model can decrease significantly and the parameters of the resulting model can be more stable.

The following subsections illustrate a couple of good ways to regularize your ensemble model. They involve estimating and choosing one or more new hyperparameters that control the amount of regularization. These hyperparameters can be determined by various methods, including a single validation data partition, cross validation, and information criteria.

Stacking with Adaptive LASSO

Consider a linear regression of the following form:

$$b(x) = w_1 * g_1 + \cdots + w_L * g_L$$

A LASSO learner finds the model weights by placing an L_1 (sum of the absolute value of the weights) penalty on the model weights as follows:

$$\min \sum_{i=1}^{N} (y_i - (w_1 * g_{1i} + \cdots + w_L * g_{Li}))^2$$

$$\text{subject to } \sum_{i=1}^{L} |w_i| \leq t$$

If the LASSO hyperparameter *t* is small enough, some of the weights will be exactly 0. Thus, the LASSO method produces a sparser and potentially more interpretable model. Adaptive LASSO (Zou 2006) modifies the LASSO penalty by applying adaptive weights (v_j) to each parameter that forms the LASSO constraint:

$$\text{subject to } \sum_{i=1}^{L}(v_i|w_i|) \leq t$$

These constraints control shrinking the zero coefficients more than they control shrinking the nonzero coefficients.

The following REGSELECT procedure run builds an adaptive LASSO model. By default, the procedure uses the inverse of the full linear regression model coefficients for v_j (Güneş 2015).

```
proc regselect data=cas.train_mean_oofs;
    partition fraction(validate=0.3);
    model target = mean_factmac mean_gbt mean_logit mean_frst / noint;
    selection method=lasso
          (adaptive stop=sbc choose=validate) details=steps;
    code file="/c/output/lasso_score.sas";
run;
```

The PARTITION statement reserves 30% of the data for validation, leaving the remaining 70% for training. The validation part of the data is used to find the optimal value for the adaptive LASSO parameter *t*. The MODEL statement specifies the four average OOF predictions from Level 2 as input variables. The SELECTION statement requests the adaptive LASSO method, and the CHOOSE=VALIDATE suboption requests that the selected regularization parameter (*t*) be used to minimize the validation error on the 30% single holdout set. The CODE statement saves the resulting scoring code in the specified directory.

Figure 2 shows the results. The gradient boosted predictor receives around 94% of the weight in the resulting ensemble, with the remaining 6% going to the forest model, along with just a little contribution from factorization machines. The ASE appears to have improved a little, but keep in mind that these results are on a new 30% holdout.

Parameter Estimates

Parameter	DF	Estimate
mean_factmac	1	0.001931
mean_gbt	1	0.938671
mean_frst	1	0.058570

ASE (Train)	0.09214
ASE (Validate)	0.09263

Figure 2. Parameter Estimates and Fit statistics for the Adaptive LASSO Stacking Model

To obtain a better measure of prediction error, you can check the ASE of the resulting model for the test set. The following SAS statements first score for the test set by using the saved score code, lasso_score.sas, and then calculate the ASE:

```
data cas.lasso_score;
    set cas.test_preds;
    %include '/c/output/lasso_score.sas';
run;

data cas.lasso_score;
    se=(p_target-target)*(p_target-target);
run;

proc cas;
    summary/ table={name='lasso_score', vars={'se'}};
run;
quit;
```

The **summary** CAS action outputs the test ASE of the adaptive LASSO ensemble model as 0.09269, which improves slightly on the average gradient boosting model, whose test ASE is 0.09273.

Stacking with Nonnegative Weights Regularization

Another regularization technique that is commonly used to build a stacked regression model is to restrict the regression coefficients to be nonnegative while performing regression. Breiman (1995) shows that when the regression coefficients are constrained to be nonnegative, the resulting ensemble models exhibit better prediction error than any of the individual models in the library. Because each model takes a nonnegative weight, the resulting ensemble model can also be interpreted more easily. The paper also shows that the additional commonly used restriction $\Sigma\, w_i = 1$ does not further improve the prediction accuracy, which is consistent with the findings here for the Adult data. A linear regression model that places nonnegative weights on a squared error loss function has the following form:

$$\min \sum_{i=1}^{N} (y_i - (w_1 * g_{1i} + \cdots + w_L * g_{Li}))^2$$

$$\text{subject t o } w_i > 0, \qquad \text{for i} = 1, \ldots, \text{L}$$

The following CQLIM procedure statements from SAS® Econometrics fit a linear least squares regression model with nonnegativity constraints on the regression weights:

```
proc cqlim data=cas.train_mean_oofs;
    model target= mean_gbt mean_frst mean_logit mean_factmac;
    restrict mean_gbt>0;
    restrict mean_frst>0;
    restrict mean_logit>0;
    restrict mean_factmac>0;
    output out=cas.cqlim_preds xbeta copyvar=target;
    ods output ParameterEstimates=paramests;
run;
```

Figure 3 shows the “Parameter Estimates” table that is generated by the CQLIM procedure. The Estimate column shows the regression weights of the stacked nonnegative least squares model for each of the four models. Here factorization machines have a slightly larger weight than in the previous adaptive LASSO model.

Parameter Estimates					
Parameter	DF	Estimate	Standard Error	t Value	Approx Pr > |t|
mean_gbt	1	0.921430	0.024091	38.25	<.0001
mean_factmac	1	0.022345	0.012701	1.76	0.0785
mean_frst	1	0.056071	0.025714	2.18	0.0292
mean_logit	1	1.0536712E-8	0	.	.

Figure 3. Regression Weights for the Nonnegative Least Squares Stacking Model

This stacked model produces a training error of 0.09228 and a testing error of 0.09269, which provides an improvement over the single best Level-2 model: the average gradient boosting model, which has a training ASE of 0.09236 and a testing ASE of 0.09273.

STACKING WITH THE SCALABLE GRADIENT BOOSTING ALGORITHM AND AUTOTUNING

Model stacking is not limited to basic models such as linear regression; any supervised learning algorithm can be used as a higher-level learning algorithm as long as it helps boost the prediction accuracy. In fact, nonlinear algorithms such as boosted trees and neural networks have been successfully used as a second- and third-level modeling algorithms in winning methods of various data science competitions.

The GRADBOOST procedure in SAS Visual Data Mining and Machine Learning fits a scalable gradient boosting model that is based that is on the boosting method described in Hastie, Tibshirani, and Friedman (2001), and its functionality is comparable to the popular **xgboost** program. PROC GRADBOOST is computationally efficient and uses fewer resources than the Gradient Boosting node in SAS Enterprise Miner uses.

The following GRADBOOST procedure run trains a stacked ensemble model by using the Level-2 OOF predictions of the four average models:

```
proc gradboost data=cas.train_mean_oofs outmodel=cas.gbt_ensemble;
    target target / level=nominal;
    input mean_factmac mean_gbt mean_logit mean_frst / level=interval;
    autotune tuningparameters=(ntrees samplingrate vars_to_try(init=4)
          learningrate(ub=0.3) lasso ridge) searchmethod=random
          samplesize=200 objective=ase kfold=5;
    ods output FitStatistics=Work._Gradboost_FitStats_
               VariableImportance=Work._Gradboost_VarImp_;
run;
```

The OUTMODEL option in the PROC statement saves the resulting trained model as a CAS table called gbt _ensemble. This table is used later for scoring the test data. The TARGET statement specifies the binary target variable, and the INPUT statement specifies the average OOF predictions that are obtained from Level-2 average models for gradient boosting, forest, logistic regression, and factorization machines.

The AUTOTUNE statement performs an automatic search for the optimal hyperparameter settings of the gradient boosting algorithm. It specifies a random search among 200 randomly selected hyperparameter settings of the gradient boosting algorithm. For assessing the resulting models, five-fold cross validation is used with the ASE metric that is specified by the following suboptions of the AUTOTUNE statement: OBJECTIVE=ASE KFOLD=5. The AUTOTUNE statement performs a search for the following parameters of the gradient boosting algorithm: number of iterations, sampling proportion, number of variables to try,

learning rate, and LASSO and ridge regularization parameters. For other parameters, the procedure uses default values (the maximum depth of a tree is 5, the maximum number of observations for a leaf is 5, and the maximum number of branches for a node is 2), but these values can also be optionally tuned. To further control the parameter search process, you can specify upper bounds, lower bounds, and initial values for the hyperparameters. The preceding statements specify an upper bound for the learning rate parameter, LEARNINGRATE (UB=0.2), and an initial value for the number of variables to try, VARS_TO_TRY (INIT=4).

Figure 4 summarizes the autotuning options that are specified in the AUTOTUNE statement.

Tuner Information	
Model Type	Gradient Boosting Tree
Tuner Objective Function	Average Squared Error
Search Method	RANDOM
Maximum Evaluations	201
Sample Size	200
Maximum Tuning Time in Seconds	36000
Validation Type	Cross-Validation
Num Folds in Cross-Validation	5
Log Level	2
Seed	1669463436

Figure 4. Autotuning Information Table

Figure 5 shows the resulting best configuration hyperparameter values.

Best Configuration	
Evaluation	86
Number of Trees	56
Number of Variables to Try	3
Learning Rate	0.10990335
Sampling Rate	0.75938235
Lasso	3.25403452
Ridge	3.64367127
Average Squared Error	0.09

Figure 5. Autotuning Best Hyperparameter Settings for the Stacking Gradient Boosting Model

The “Tuner Summary” table in Figure 6 shows that the five-fold ASE for the best configuration of hyperparameter values is 0.09245.

Tuner Summary	
Initial Configuration Objective Value	0.09330
Best Configuration Objective Value	0.09245
Worst Configuration Objective Value	0.1448
Initial Configuration Evaluation Time in Seconds	10.4112
Best Configuration Evaluation Time in Seconds	9.2367
Number of Improved Configurations	5
Number of Evaluated Configurations	201
Total Tuning Time in Seconds	300.98
Parallel Tuning Speedup	10.1676

Figure 6. Autotuning Summary Table for the Stacking Gradient Boosting Model

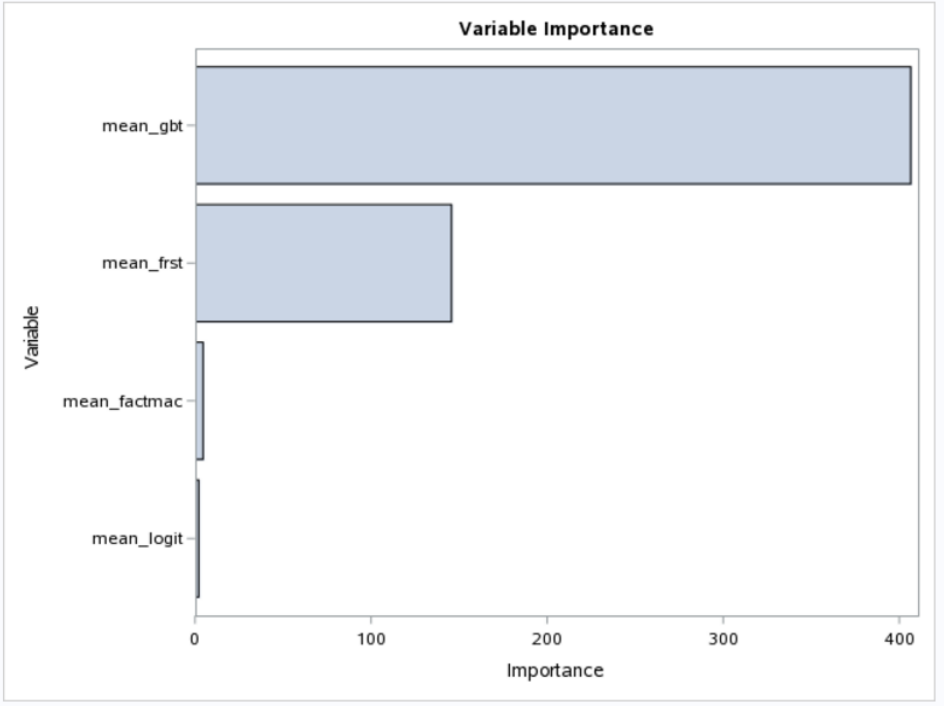
Figure 6 also reports the total tuning time to be 5 minutes. This time is based on using 100 nodes in a SAS Viya distributed analytics platform. Note that five-fold cross validation is used as an assessment measure and models are assessed for 200 different hyperparameter settings, which requires fitting and scoring for 1,000 models. Each training set includes approximately 25,600 samples (4/5 of the full training set) and 4 features, and training and scoring for one model took around 0.35 seconds. This brief amount of time is made possible by taking full advantage of in-memory parallel computing not only for running each gradient boosting model but also for performing a random search for hyperparameter tuning.

The output also includes a table of the parameter settings and the corresponding five-fold ASEs for all 200 hyperparameter settings. Figure 7 shows the best 10 models that are found by the autotuning functionality. The AUTOTUNE statement in SAS Viya machine learning procedures has even more extensive capabilities that are not covered here; for more information and full capabilities, see Koch et al. (2017).

Tuner Results Default and Best Configurations							
Evaluation	Number of Trees	Number of Variables to Try	Learning Rate	Sampling Rate	Lasso	Ridge	Average Squared Error
0	100	4	0.100000	0.500000	0	0	0.0933
86	56	3	0.109903	0.759382	3.254035	3.643671	0.0924
78	84	3	0.078068	0.618316	7.852888	3.410856	0.0925
17	93	2	0.074230	0.545497	2.438973	1.077369	0.0925
149	76	3	0.149835	0.991542	6.293911	3.741891	0.0926
37	61	3	0.096917	0.574695	3.551582	2.478200	0.0926
108	76	4	0.066486	0.681446	8.874643	6.520876	0.0926
117	52	2	0.191996	0.959816	0.214059	5.811560	0.0926
40	54	3	0.109288	0.778913	9.021802	0.708567	0.0926
129	65	3	0.069800	0.851765	5.150564	0.671172	0.0926
126	50	3	0.104418	0.808189	9.780431	0.641345	0.0926

Figure 7. Autotuning Results for the Best 10 Models

Figure 8 plots the variable importance for the selected hyperparameter settings of the gradient boosting model. The two tree-based methods dominate.



The following PROC GRADBOOST statements use the saved stacked ensemble model (cas.gbt_ensemble) to score the test set. The input data set (cas.test_mean_preds) includes Level-2 predictions for the test set.

```
proc gradboost data=cas.test_mean_preds inmodel=cas.gbt_ensemble;
    output out=cas.test_gbtscr copyvars=(id target);
run;
```

The following SAS code calculates the test ASE for the gradient boosting stacked ensemble model for the test data:

```
data cas.test_gbtscr;
    se=(p_target1-target)*(p_target1-target);
run;

proc cas;
    summary/ table={name='test_gbtscr', vars={'se'}};
run;
quit;
```

The **summary** action reports the ASE of the test data as 0.09298.

PROC GRADBOOST runs the **gbtreetrain** and **gbtreescore** CAS actions (in the Decision Tree action set) behind the scenes to train and score gradient boosting models. Appendix A provides a step-by-step CAS language (CASL) program that uses these actions to find the five-fold OOF predictions and cross validation ASE of the model for the hyperparameter values that are found here. Programming through CASL and CAS actions often requires more coding compared to using packaged machine learning

procedures, which are essentially bundled functions of CAS actions. However, programming this way offers you more flexibility and control over the whole model building process. You can also call CAS actions through other languages such as Python and Lua.

HILL CLIMBING

The hill climbing technique (Caruana et al. 2004) is similar to forward stepwise selection. At each step of the selection, the model in the model library that maximizes the preferred performance metric joins the ensemble, and the ensemble is updated to be a simple weighted average of models. Hill climbing differs from regular stepwise selection in that rather than fitting a linear model at each step of the selection, it adds models to an ensemble by averaging predictions with the models already in the ensemble. As such, it is actually a form of nonnegative least squares, because the coefficients of each model are guaranteed to be nonnegative. Building an ensemble model this way can be very efficient computationally and has the significant advantage of being readily applicable to any performance metric of interest.

Caruana et al. (2004) use a hill climbing (single holdout validation) set at each step of the selection process to assess model performance. A separate validation set plays an important role in order to deal with overfitting, especially when you use the regular training predictions as input variables. However, instead of using a hill climbing validation set, this paper's analysis performs hill climbing on the library of OOF predictions. This approach deals with overfitting while maximally using the training data for the critical hill climbing step.

At each iteration of the hill climbing algorithm, every candidate model is evaluated to find the one that maximally improves the ensemble in a greedy fashion. Selection with replacement allows models to be added to the ensemble multiple times, permitting an already used model to be selected again rather than adding an unused model (which could possibly hurt the ensemble model's performance). Thus each model in the ensemble model can take different weights based on how many times it is selected.

For the Adult data, an ensemble model is built by using hill climbing for combining the four average Level-2 models. Figure 8 shows that the first model to enter the ensemble is the single best gradient boosted tree (gbt) model with a five-fold training cross validation ASE of 0.09235. Hill climbing keeps adding the same gradient boosting model until step 7. At step 7, the forest model joins the ensemble, which helps decrease both the training and testing errors nicely.

Obs	Step	Model	ASE_test	ASE_train
1	1	mean_gbt	0.092734	0.09235
2	2	mean_gbt	0.092734	0.09235
3	3	mean_gbt	0.092734	0.09235
4	4	mean_gbt	0.092734	0.09235
5	5	mean_gbt	0.092734	0.09235
6	6	mean_gbt	0.092734	0.09235
7	7	mean_frst	0.092684	0.09235
8	8	mean_gbt	0.092679	0.09234
9	9	mean_gbt	0.092678	0.09233
10	10	mean_gbt	0.092678	0.09233

Obs	Step	Model	ASE_test	ASE_train
10	10	mean_gbt	0.092678	0.09233
11	11	mean_gbt	0.092679	0.09233
12	12	mean_gbt	0.092680	0.09233
13	13	mean_gbt	0.092682	0.09233
14	14	mean_gbt	0.092684	0.09233
15	15	mean_gbt	0.092685	0.09233
16	16	mean_gbt	0.092687	0.09233
17	17	mean_gbt	0.092689	0.09233
18	18	mean_gbt	0.092690	0.09233
19	19	mean_gbt	0.092692	0.09233
20	20	mean_gbt	0.092693	0.09233

Figure 8. First 20 Steps of Hill Climbing

Figure 9 shows graphically how the training and test errors change by the hill climbing steps. It shows that after step 9, the training error does not change much, but the test error increases slightly. The model at

step 9 has a training ASE of 0.09268 and a testing ASE of 0.09233. If you choose this model at step9 as the final hill climbing model, the Level-2 average gradient boosting model takes a weight of 8, the Level-2 average forest model takes a weight of 1, and the other two models take 0 weights. In a typical hill climbing ensemble model, it is common to see powerful models being selected multiple times. In this case, the gbt model dominates but is complemented by a small contribution from forest model.

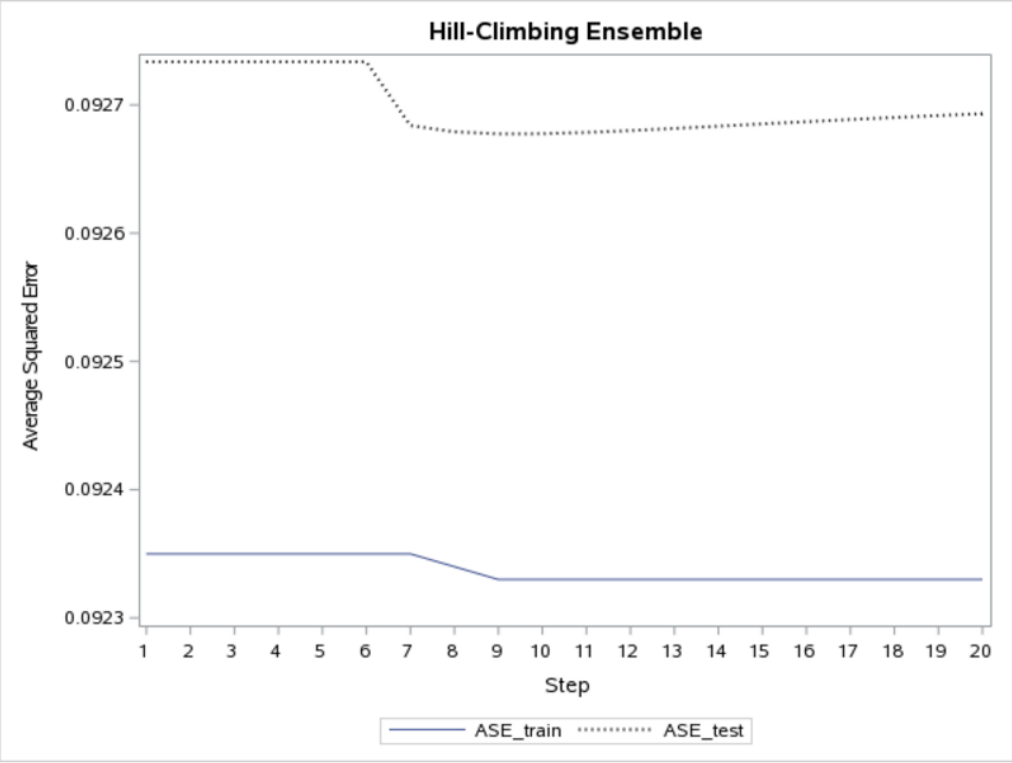


Figure 9. First 20 Steps of Hill Climbing with the Corresponding Training and Test ASEs

Because the hill climbing SAS program is lengthy, it is not provided here. See the GitHub repository for the full hill climbing program, which is written in the CAS language. The program is very flexible, and you can run it in the SAS Viya environment to build your own hill climbing ensemble model for your data.

When the same objective function is used, the nonnegative least squares approach is a generalization of hill climbing technique. For this example, Figure 10 shows that all three Level-3 linear modeling approaches (adaptive LASSO, nonnegative least squares, and hill climbing) produced very similar results and decreased the test ASE when compared to the single best model of Level-2 (shown in last row). On the other hand, the Level-3 stacked gradient boosting model did not provide a better model than the Level-2 average gradient boosting model.

Note that since the adaptive LASSO and the nonnegative least squares models weights are so close to each other, the training and test ASEs are almost the same when five decimal points are used. Note also that training ASEs are calculated when the Level-3 models are fit on the full training data.

Models	**Training ASE**	**Test ASE**
Level-3 adaptive LASSO	0.09269	0.09228
Level-3 nonnegative least squares	0.09269	0.09228
Level-3 gradient boosting	0.09130	0.09298
Level-3 hill climbing	0.09268	0.09233
Level-2 best model: average gradient boosting	**0.09236**	**0.09273**

Figure 10. Level-3 Stacked Models Training and Test ASEs Compared to the Single Best Level-2 Model

CONCLUSION

Stacked ensembling is an essential tool in any expert data scientist's toolbox. This paper shows how you can perform this valuable technique in the new SAS Viya framework by taking advantage of powerful underlying machine learning algorithms that are available through CAS procedures and actions.

REFERENCES

Breiman, L. 1995. "Stacked Regressions." *Machine Learning* 24:49–64.

Caruana, R., Niculescu-Mizil, A., Crew, G., and Ksikes, A. 2004. "Ensemble Selection from Libraries of Models." *Proceedings of the Twenty-First International Conference on Machine Learning*. New York: ACM.

Cawley, G. C., and Talbot, N. L. 2010. "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation." *Journal of Machine Learning Research* 11:2079–2107.

Dietterich, T. 2000. "Ensemble Methods in Machine Learning." *Lecture Notes in Computer Science* 1857:1–15.

Güneş, F. 2015. "Penalized Regression Methods for Linear Models in SAS/STAT." Cary, NC: SAS Institute Inc. Available at https://support.sas.com/rnd/app/stat/papers/2015/PenalizedRegression_LinearModels.pdf.

Hastie, T. J., Tibshirani, R. J., and Friedman, J. H. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* New York: Springer-Verlag.

Koch, P., Wujek, B., Golovidov, O., and Gardner, S. 2017. "Automated Hyperparameter Tuning for Effective Machine Learning." In *Proceedings of the SAS Global Forum 2017 Conference.* Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/ proceedings17/SAS514-2017.pdf.

Lichman, M. 2013. UCI Machine Learning Repository. School of Information and Computer Sciences, University of California, Irvine. Available at http://archive.ics.uci.edu/ml.

Sill, J., Takacs, G., Mackey, L., and Lin, D. 2009. "Feature-Weighted Linear Stacking." CoRR abs/0911.0460.

Tibshirani, R. 1996. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society*, Series B 58:267–288.

Van der Laan, M. J., Polley, E. C., and Hubbard, A. E. 2007. "Super Learner." *U.C. Berkeley Division of Biostatistics Working Paper Series*, Working Paper 222.

Wexler, J., Haller, S., and Myneni, R. 2017. "An Overview of SAS Visual Data Mining and Machine Learning on SAS Viya." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/ proceedings17/SAS1492-2017.pdf.

Wujek, B., Hall, P., and Güneş, F. 2016. "Best Practices in Machine Learning Applications." In *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings16/SAS2360-2016.pdf.

Zou, H. 2006. "The Adaptive Lasso and Its Oracle Properties." *Journal of the American Statistical Association* 101:1418–1429.

ACKNOWLEDGMENTS

The authors are grateful to Wendy Czika and Padraic Neville of the Advanced Analytics Division of SAS for helpful comments and support. The authors also thank Anne Baxter for editorial assistance.

RECOMMENDED READING

A GitHub repository is available at https://github.com/sassoftware/sas-viya-machine-learning/stacking.The repository contains several different programs to help you reproduce results in this paper. The repository also contains supplemental material, including a detailed breakdown of some additional ensembling that is performed using Level-2 bootstrap samples.

Getting Started with SAS® Visual Data Mining and Machine Learning

SAS® Visual Data Mining and Machine Learning : Data Mining and Machine Learning Procedures

SAS® Visual Data Mining and Machine Learning : Statistical Procedures

SAS® Econometrics: Econometrical Procedures

SAS® Visual Data Mining and Machine Learning : Data Mining and Machine Learning Programming Guide

SAS® Cloud Analytic Services: CAS Procedure Programming Guide and Reference

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Funda Güneş	Russ Wolfinger	Pei-Yi Tan
SAS Institute Inc.	SAS Institute Inc.	SAS Institute Inc.
funda.gunes@sas.com	Russ.Wolfinger@jmp.com	Pei-Yi.Tan@sas.com

APPENDIX A:

This appendix provides a step-by-step CAS language program that calculates and saves five-fold OOF predictions of the stacked ensemble model with the scalable gradient boosting algorithm for the set of hyperparameters that are shown in Figure 5. You can easily modify this program to obtain OOF predictions for your models that might use different machine learning training and scoring CAS actions.

```
/* Start a CAS session named mySession */
cas mySession;

/* Define a CAS engine libref for CAS in-memory data tables */
libname cas sasioca;

/* Create a SAS libref for the directory that has the data */
libname data "/folders/myfolders/";

/* Load OOF predictions into CAS using a DATA step */
data cas.train_oofs;
      set data.train_oofs;
      _fold_=int(ranuni(1)*5)+1;
run;

proc cas;
      /* Create an input variable list for modeling*/
      input_vars={{name='mean_gbt'},{name='mean_frst'},{name='mean_logit'},
                  {name='mean_factmac'}};
      nFold=5;
```

```
      do i=1 to nFold;
            /* Generate no_fold_i and fold_i variables */
            no_fold_i = "_fold_ ne " || (String)i;
            fold_i    = "_fold_ eq " || (String)i;

            /* Generate a model name to store the ith trained model */
            mymodel = "gbt_" || (String)i;

            /* Generate a cas table name to store the scored data */
            scored_data = "gbtscore_" || (String)i;

            /* Train a gradient boosting model without fold i */
            decisiontree.gbtreetrain result=r1 /
                  table={name='train_mean_oofs',  where=no_fold_i}
                  inputs=input_vars
                  target="target"
                  maxbranch=2
                  maxlevel=5
                  leafsize=60
                  ntree=56
                  m=3
                  binorder=1
                  nbins=100
                  seed=1234
                  subsamplerate=0.75938
                  learningRate=0.10990
                  lasso=3.25403
                  ridge=3.64367
                  casout={name=mymodel, replace=1};
                  print r1;

            /* Score for the left out fold i */
            decisionTree.gbtreescore result = r2/
                  table={name='train_mean_oofs', where=fold_i}
                  model={name=mymodel}
                  casout={name=scored_data, replace=TRUE }
                  copyVars={"id", "target"}
                  encodeName=true;
      end;
   quit;

   /* Put together OOF predictions */
   data cas.gbt_stack_oofs (keep= id target p_target se);
      set cas.gbtscore_1-cas.gbtscore_5;
      se=(p_target-target)*(p_target-target);
      run;
   run;

   /* The mean value for variable se is the 5-fold cross validation error */
   proc cas;
      summary / table={name='gbt_stack_oofs', vars={'se'}};
   run;
   /* Quit PROC CAS */
quit;
```