SAS514-2017

Automated Hyperparameter Tuning for Effective Machine Learning

Patrick Koch, Brett Wujek, Oleg Golovidov, and Steven Gardner

SAS Institute Inc.

ABSTRACT

Machine learning predictive modeling algorithms are governed by “hyperparameters” that have no clear defaults agreeable to a wide range of applications. The *depth* of a decision tree, *number of trees* in a forest, *number of hidden layers* and *neurons in each layer* in a neural network, and *degree of regularization* to prevent overfitting are a few examples of quantities that must be prescribed for these algorithms. Not only do ideal settings for the hyperparameters dictate the performance of the training process, but more importantly they govern the quality of the resulting predictive models. Recent efforts to move from a manual or random adjustment of these parameters include rough grid search and intelligent numerical optimization strategies.

This paper presents an automatic tuning implementation that uses local search optimization for tuning hyperparameters of modeling algorithms in SAS® Visual Data Mining and Machine Learning. The AUTOTUNE statement in the TREESPLIT, FOREST, GRADBOOST, NNET, SVMACHINE, and FACTMAC procedures defines tunable parameters, default ranges, user overrides, and validation schemes to avoid overfitting. Given the inherent expense of training numerous candidate models, the paper addresses efficient distributed and parallel paradigms for training and tuning models on the SAS® Viya™ platform. It also presents sample tuning results that demonstrate improved model accuracy and offers recommendations for efficient and effective model tuning.

INTRODUCTION

Machine learning is a form of self-calibration of predictive models that are built from training data. Machine learning predictive modeling algorithms are commonly used to find hidden value in big data. Facilitating effective decision making requires the transformation of relevant data to high-quality descriptive and predictive models. The transformation presents several challenges however. For example, consider a neural network, as shown in Figure 1. Outputs are predicted by transforming a set of inputs through a series of hidden layers that are defined by activation functions linked with weights. Determining the activation functions and the weights to determine the best model configuration is a complex optimization problem.

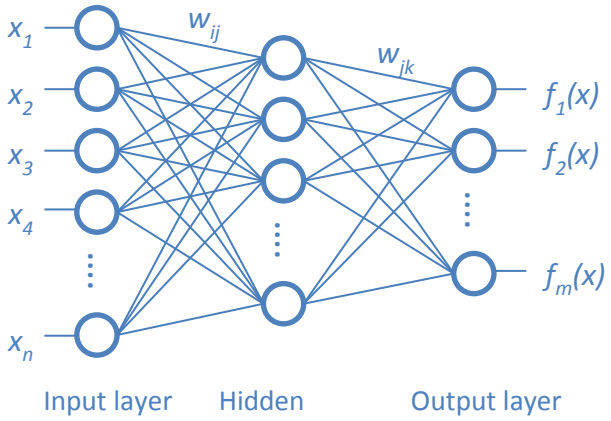
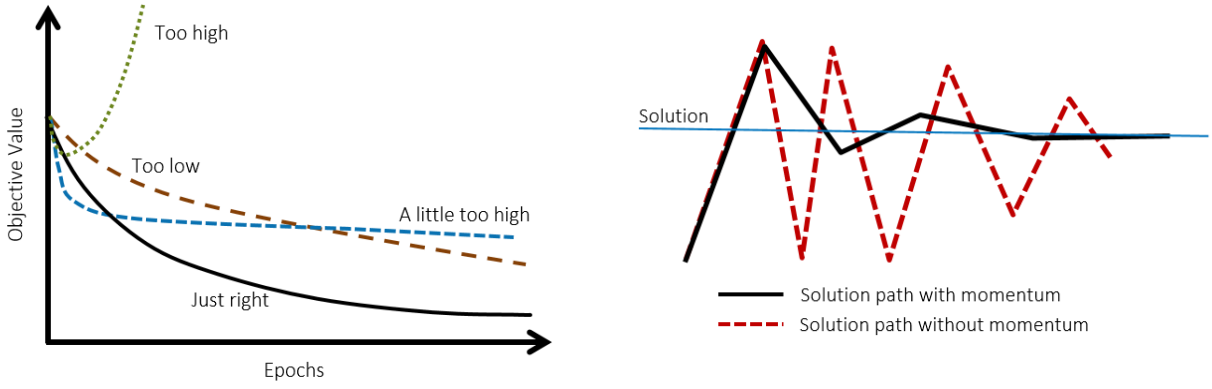


Figure 1. Neural Network

The goal in this model-training optimization problem is to find the weights that will minimize the error in model predictions based on the training data, validation data, specified model configuration (number of hidden layers and number of neurons in each hidden layer), and the level of regularization that is added to reduce overfitting to training data. One recently popular approach to solving for the weights in this optimization problem is through use of a *stochastic gradient descent* (SGD) algorithm (Bottou, Curtis, and Nocedal 2016). This algorithm is a variation of gradient descent in which instead of calculating the gradient of the loss over all observations to update the weights at each step, a "mini-batch" random sample of the observations is used to estimate loss, sampling without replacement until all observations have been used. The performance of this algorithm, as with all optimization algorithms, depends on a number of control parameters for which no default values are best for all problems. SGD parameters include the following control parameters (among others):

- a *learning rate* that controls the step size for selecting new weights
- a *momentum* parameter to avoid slow oscillations
- an *adaptive decay rate* and an *annealing rate* to adjust the learning rate for each weight and time
- a *mini-batch* size for sampling a subset of observations

The best values of the control parameters must be chosen very carefully. For example, the learning rate can be adjusted to reach a solution more quickly; however, if the value is too high, the solution diverges, and if it is too low, the performance is very slow, as shown in Figure 2(a). The momentum parameter dictates whether the algorithm tends to oscillate slowly in ravines where solutions lie (jumping back and forth across the ravine) or dives in quickly, as shown in Figure 2(b). But if momentum is too high, it could jump past the solution (Sutskever et al. 2013). Similar accuracy-versus-performance trade-offs are encountered with the other control parameters. The adaptive decay can be adjusted to improve accuracy, and the annealing rate is often necessary to avoid jumping past a solution. Ideally, the size of the mini-batch for distributed training is small enough to improve performance and large enough to produce accurate models. A communication frequency parameter can be used to adjust how often training information (such as average weights, velocity vectors, and annealing rates) is synced when training is distributed across a compute grid; higher frequency might increase accuracy, but it also reduces performance.



(a) Learning Rate **(b) Momentum**

Figure 2. Effect of Hyperparameters on Neural Network Training Convergence

The best values of these parameters vary for different data sets, and they must be chosen before model training begins. These options dictate not only the performance of the training process, but more importantly, the quality of the resulting model. Because these parameters are external to the training

process—that is, they are not the model parameters (weights in the neural network) being optimized during training—they are often called *hyperparameters*. Figure 3 depicts the distinction between *training* a model (solving for model parameters) and *tuning* a model (finding the best algorithm hyperparameter values). Settings for these hyperparameters can significantly influence the resulting accuracy of the predictive models, and there are no clear defaults that work well for different data sets. In addition to the optimization options already discussed for the SGD algorithm, the machine learning algorithms themselves have many hyperparameters. As in the neural network example, the *number of hidden layers*, the *number of neurons in each hidden layer*, the *distribution used for the initial weights*, and so on are all hyperparameters that are specified up front for model training, that govern the quality of the resulting model, and whose ideal values also vary widely with different data sets.

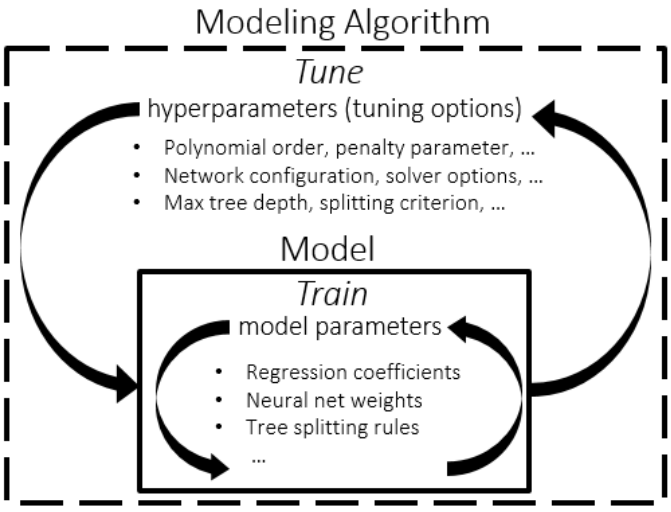


Figure 3. Model Training in Relation to Model Tuning

Tuning hyperparameter values is a critical aspect of the model training process and is considered a best practice for a successful machine learning application (Wujek, Hall, and Güneş 2016). The remainder of this paper describes some of the common traditional approaches to hyperparameter tuning and introduces a new hybrid approach in SAS Visual Data Mining and Machine Learning that takes advantage of the combination of the powerful machine learning algorithms, optimization routines, and distributed and parallel computing that running on the SAS Viya platform offers.

HYPERPARAMETER TUNING

The approach to finding the ideal values for hyperparameters (tuning a model to a particular data set) has traditionally been a manual effort. For guidance in setting these values, researchers often rely on their past experience using these machine learning algorithms to train models. However, even with expertise in machine learning algorithms and their hyperparameters, the best settings of these hyperparameters will change with different data; it is difficult to prescribe the hyperparameter values based on previous experience. The ability to explore alternative configurations in a more guided and automated manner is needed.

COMMON APPROACHES

Grid Search

A typical approach to exploring alternative model configurations is by using what is commonly known as a grid search. Each hyperparameter of interest is discretized into a desired set of values to be studied, and

models are trained and assessed for all combinations of the values across all hyperparameters (that is, a "grid"). Although fairly simple and straightforward to carry out, a grid search is quite costly because expense grows exponentially with the number of hyperparameters and the number of discrete levels of each. Even with the inherent ability of a grid search to train and assess all candidate models in parallel (assuming an appropriate environment in which to do so), it must be quite coarse in order to be feasible, and thus it will often fail to identify an improved model configuration. Figure 4(a) illustrates hypothetical distributions of two hyperparameters (X_1 and X_2) with respect to a training objective and depicts the difficulty of finding a good combination with a coarse standard grid search.

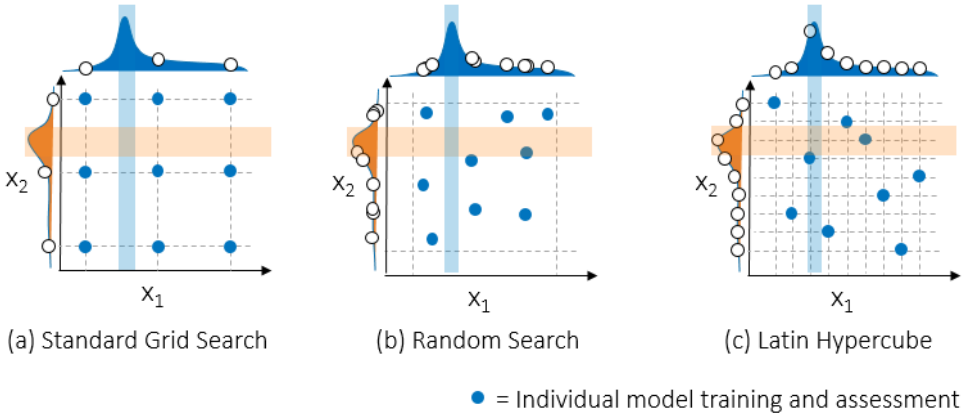


Figure 4. Common Approaches to Hyperparameter Tuning

Random Search

A simple yet surprisingly effective alternative to performing a grid search is to train and assess candidate models by using random combinations of hyperparameter values. As demonstrated in Bergstra and Bengio (2012), given the disparity in the sensitivity of model accuracy to different hyperparameters, a set of candidates that incorporates a larger number of trial values for each hyperparameter will have a much greater chance of finding effective values for each hyperparameter. Because some of the hyperparameters might actually have little to no effect on the model for certain data sets, it is prudent to avoid wasting the effort to evaluate all combinations, especially for higher-dimensional hyperparameter spaces. Rather than focusing on studying a full-factorial combination of all hyperparameter values, studying random combinations enables you to explore more values of each hyperparameter at the same cost (the number of candidate models that are trained and assessed). Figure 4(b) depicts a potential random distribution with the same budget of evaluations (nine points in this example) as shown for the grid search in Figure 4(a), highlighting the potential to find better hyperparameter values. Still, the effectiveness of evaluating purely random combinations of hyperparameter values is subject to the size and uniformity of the sample; candidate combinations can be concentrated in regions that completely omit the most effective values of one or more of the hyperparameters.

Latin Hypercube Sampling

A similar but more structured approach is to use a random Latin hypercube sample (LHS) (McKay 1992), an experimental design in which samples are exactly uniform across each hyperparameter but random in combinations. These so-called low-discrepancy point sets attempt to ensure that points are approximately equidistant from one another in order to fill the space efficiently. This sampling allows for coverage across the entire range of each hyperparameter and is more likely to find good values of each hyperparameter—as shown in Figure 4(c)—which can then be used to identify good combinations. Other experimental design procedures can also be quite effective at ensuring equal density sampling throughout the entire hyperparameter space, including optimal Latin hypercube sampling as proposed by Sacks et al. (1989).

Optimization

Exploring alternative model configurations by evaluating a discrete sample of hyperparameter combinations, whether randomly chosen or through a more structured experimental design approach, is certainly a fairly straightforward approach. However, true hyperparameter optimization should allow the use of logic and information from previously evaluated configurations to determine how to effectively search through the space. Discrete samples are unlikely to identify even a local accuracy peak or error valley in the hyperparameter space; searching between these discrete samples can uncover good combinations of hyperparameter values. The search is based on an objective of minimizing the model validation error, so each "evaluation" from the optimization algorithm's perspective is a full cycle of model training and validation. These methods are designed to make intelligent use of fewer evaluations and thus save on the overall computation time. Optimization algorithms that have been used for hyperparameter tuning include Broyden-Fletcher-Goldfarb-Shanno (BFGS) (Konen et al. 2011), covariance matrix adaptation evolution strategy (CMA-ES) (Konen et al. 2011), particle swarm (PS) (Renukadevi and Thangaraj 2014; Gomes et al. 2012), tabu search (TS) (Gomes et al. 2012), genetic algorithms (GA) (Lorena and de Carvalho 2008), and more recently surrogate-based Bayesian optimization (Denwancker et al. 2016).

However, because machine learning training and scoring algorithms are a complex black box to the tuning algorithm, they create a challenging class of optimization problems. Figure 5 illustrates several of these challenges:

- Machine learning algorithms typically include not only continuous variables but also categorical and integer variables. These variables can lead to very discrete changes in the objective.
- In some cases, the hyperparameter space is discontinuous and the objective evaluation fails.
- The space can also be very noisy and nondeterministic (for example, when distributed data are moved around because of unexpected rebalancing).
- Objective evaluations can fail because a compute node fails, which can derail a search strategy.
- Often the space contains many flat regions where many configurations produce very similar models.

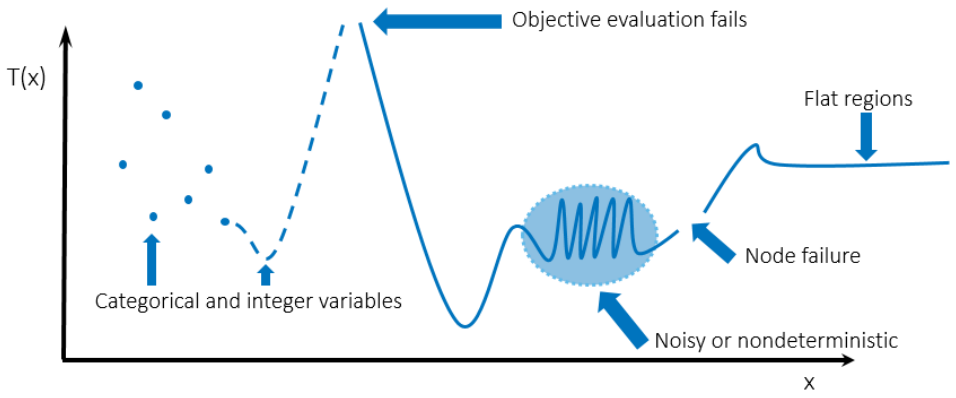


Figure 5. Challenges in Applying Optimization to Hyperparameter Tuning

An additional challenge is the unpredictable computation expense of training and validating predictive models using different hyperparameter values. For example, adding hidden layers and neurons to a neural network can significantly increase the training and validation time, resulting in widely ranging potential objective expense. Given the great promise of using intelligent optimization techniques coupled with the aforementioned challenges of applying these techniques for tuning machine learning hyperparameters, a very flexible and efficient search strategy is needed.

AUTOTUNING ON THE SAS VIYA PLATFORM

SAS Viya is a new platform that enables parallel/distributed computing of the powerful analytics that SAS provides. The new SAS Visual Data Mining and Machine Learning offering (Wexler, Haller, and Myneni 2017) provides a hyperparameter autotuning capability that is built on local search optimization in SAS® software. Optimization for hyperparameter tuning typically can very quickly reduce, by several percentage points, the model error that is produced by default settings of these hyperparameters. More advanced and extensive optimization, facilitated through parallel tuning to explore more configurations and refine hyperparameter values, can lead to further improvement. With increased dimensionality of the hyperparameter space (that is, as more hyperparameters require tuning), a manual tuning process becomes much more difficult and a much coarser grid search is required. An automated, parallelized search strategy can also benefit novice machine learning algorithm users.

LOCAL SEARCH OPTIMIZATION

SAS local search optimization (LSO) is a hybrid derivative-free optimization framework that operates in the SAS Viya parallel/distributed environment to overcome the challenges and expense of hyperparameter optimization. As shown in Figure 6, it consists of an extendable suite of search methods that are driven by a hybrid solver manager that controls concurrent execution of search methods. Objective evaluations (different model configurations in this case) are distributed across multiple evaluation worker nodes in a compute grid and coordinated in a feedback loop that supplies data from all concurrent running search methods. The strengths of this approach include handling of continuous, integer, and categorical variables; handling nonsmooth, discontinuous spaces; and ease of parallelizing the search strategy.

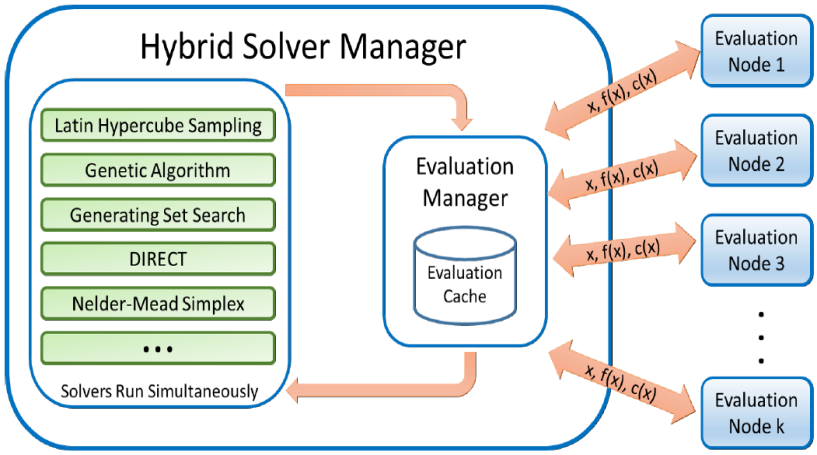


Figure 6. Local Search Optimization: Parallel Hybrid Derivative-Free Optimization Strategy

The autotuning capability in SAS Visual Data Mining and Machine Learning takes advantage of the LSO framework to provide a flexible and effective hybrid search strategy. It uses a default hybrid search strategy that begins with a Latin hypercube sample (LHS), which provides a more uniform sample of the hyperparameter space than a grid or random search provides. The best samples from the LHS are then used to seed a genetic algorithm (GA), which crosses and mutates the best samples in an iterative process to generate a new population of model configurations for each iteration. An important note here is that the LHS samples can be evaluated in parallel and the GA population at each iteration can be evaluated in parallel. Alternate search methods include a single Latin hypercube sample, a purely random sample, and an experimental Bayesian search method.

AUTOTUNING IN SAS MODELING PROCEDURES

The hybrid strategy for automatically tuning hyperparameters is used by a number of modeling procedures in SAS Visual Data Mining and Machine Learning. Any modeling procedure that supports autotuning provides an AUTOTUNE statement, which includes a number of options for specifically configuring what to tune and how to perform the tuning process. The following example shows how the simple addition of a single line (`autotune;`) to an existing GRADBOOST procedure script triggers the process of autotuning a gradient boosting model. The best found configuration of hyperparameters is reported as an ODS table, and the corresponding best model is saved in the specified data table (`mycaslib.mymodel`).

```
cas mysess;
libname mycaslib sasioca casref=mysess;

data mycaslib.dmagecr;
    set sampsio.dmagecr;
run;

proc gradboost data=mycaslib.dmagecr outmodel=mycaslib.mymodel;
    target good_bad / level=nominal;
    input checking duration history amount savings employed installp
        marital coapp resident property age other housing existcr job
        depends telephon foreign / level=interval;
    input purpose / level=nominal;
    autotune;
run;
```

Note: If your installation does not include the Sampsio library of examples, you will need to define it explicitly by running the following command:

```
libname sampsio '!sasroot/samples/samplesml';
```

After you run a modeling procedure that includes the AUTOTUNE statement, you will see (in addition to the standard ODS output that the procedure produces) the following additional ODS tables, which are produced by the autotuning algorithm:

- **Tuner Information** displays the tuner configuration.
- **Tuner Summary** summarizes tuner results, which include initial, best, and worst configuration; number of configurations; and tuning clock time and observed parallel speed up. (For more information, see the section “Autotuning Results and Recommendations.”)
- **Tuner Task Timing** displays the time that was used for training, scoring, tuner overhead, and the overall CPU time that was required.
- **Best Configuration** provides the best configuration evaluation number, final hyperparameter values, and best configuration objective value.
- **Tuner Results** displays the initial configuration as Evaluation 0 on the first row of the table, followed by up to 10 best found configurations, sorted by their objective function value. This table enables you to compare the initial and best found configurations and potentially choose a simpler model that has nearly equivalent accuracy.
- **Tuner History** displays hyperparameter and objective values for all evaluated configurations.

Figure 7 shows some of the tables that result from running the preceding SAS script. Note that random seed generation and data distribution in SAS Viya will cause results to vary.

Tuner Information	
Model Type	Gradient Boosting Tree
Tuner Objective Function	Misclassification Error Percentage
Search Method	GA
Maximum Evaluations	50
Population Size	10
Maximum Iterations	5
Maximum Tuning Time in Seconds	36000
Validation Type	Single Partition
Validation Partition Fraction	0.3
Log Level	3
Seed	325840536

Tuner Summary	
Initial Configuration Objective Value	25.2492
Best Configuration Objective Value	22.9236
Worst Configuration Objective Value	32.8904
Initial Configuration Evaluation Time in Seconds	205.86
Best Configuration Evaluation Time in Seconds	199.53
Number of Improved Configurations	2
Number of Evaluated Configurations	45
Total Tuning Time in Seconds	1337.33
Parallel Tuning Speedup	4.7757

Tuner Task Timing		
Task	Seconds	Percent
Model Training	6294.43	98.55
Model Scoring	79.06	1.24
Total Objective Evaluations	6373.49	99.79
Tuner	13.25	0.21
Total CPU Time	6386.74	100.00

Best Configuration	
Evaluation	15
Number of Trees	121
Number of Variables to Try	18
Learning Rate	0.10852878
Sampling Rate	0.71321939
Lasso	2.36910431
Ridge	1.42146259
Misclassification Error Percentage	22.92

Tuner Results
Default and Best Configurations

Evaluation	Number of Trees	Number of Variables to Try	Learning Rate	Sampling Rate	Lasso	Ridge	Misclassification Error Percentage
0	100	20	0.100000	0.500000	0	0	25.25
15	121	18	0.108529	0.713219	2.369104	1.421463	22.92
39	120	18	0.117971	0.715619	2.518662	1.505052	23.92
10	78	18	0.450000	0.800000	7.777778	4.444444	24.25
44	122	17	0.167917	0.647899	2.607746	1.579147	24.25
37	125	18	0.101324	0.679376	2.287135	1.447222	24.58
38	124	18	0.087410	0.707852	2.034603	1.234505	24.58
14	71	20	0.088529	0.213219	0.755581	1.809792	24.92
30	122	18	0.099167	0.710840	2.220824	1.338587	24.92
1	100	20	0.100000	0.500000	0	0	25.25
6	121	9	1.000000	0.100000	4.444444	2.222222	25.25

Figure 7. SAS ODS Output Tables Produced by Autotuning

For each modeling procedure that supports autotuning, the autotuning process automatically tunes a specific subset of hyperparameters. For any hyperparameter being tuned, the procedure ignores any value that is explicitly specified in a statement other than the AUTOTUNE statement; instead the

autotuning process dictates both an initial value and subsequent values for candidate model configurations, either using values or ranges that are specified in the AUTOTUNE statement or using internally prescribed defaults. Table 1 lists the hyperparameters that are tuned and their corresponding defaults for the various modeling procedures.

Hyperparameter	Initial Value	Lower Bound	Upper Bound
Decision Tree (PROC TREESPLIT)			
MAXDEPTH	10	1	19
NUMBIN	20	20	200
GROW	GAIN (nominal target)	GAIN, IGR, GINI, CHISQUARE, CHAID (nominal target)	
	VARIANCE (interval target)	VARIANCE, FTEST, CHAID (interval target)	
Forest (PROC FOREST)			
NTREES	100	20	150
VARS_TO_TRY	sqrt(# inputs)	1	# inputs
INBAGFRACTION	0.6	0.1	0.9
MAXDEPTH	20	1	29
Gradient Boosting Tree (PROC GRADBOOST)			
NTREES	100	20	150
VARS_TO_TRY	# inputs	1	# inputs
LEARNINGRATE	0.1	0.01	1.0
SAMPLINGRATE	0.5	0.1	1.0
LASSO	0.0	0.0	10.0
RIDGE	0.0	0.0	10.0
Neural Network (PROC NNET)			
NHIDDEN	0	0	5
NUNITS1,…,5	1	1	100
REGL1	0	0	10.0
REGL2	0	0	10.0
LEARNINGRATE*	1 E–3	1E–6	1 E–1
ANNEALINGRATE*	1 E–6	1E–13	1 E–2
*These hyperparameters apply only when the neural net training optimization algorithm is SGD.			
Support Vector Machine (PROC SVMACHINE)			
C	1.0	1E–10	100.0
DEGREE	1	1	3
Factorization Machine (PROC FACTMAC)			
NFACTORS	5	5, 10, 15, 20, 25, 30	
MAXITER	30	10, 20, 30, …, 200	
LEARNSTEP	1 E–3	1 E–6, 1 E–5, 1 E–4, 1 E–3, 1 E–2, 1 E–1, 1.0	

Table 1. Hyperparameters Driven by Autotuning in SAS Procedures

In addition to defining *what* to tune, you can set various options for *how* the tuning process should be carried out and when it should be terminated. The following example demonstrates how a few of these options can be added to the AUTOTUNE statement in the script shown earlier:

```
proc gradboost data=mycaslib.dmagecr outmodel=mycaslib.mymodel;
    target good_bad / level=nominal;
    input checking duration history amount savings employed installp
        marital coapp resident property age other housing existcr job
        depends telephon foreign / level=interval;
    input purpose / level=nominal;
    autotune popsize=5 maxiter=3 objective=ASE;
run;
```

Table 2 lists all the available AUTOTUNE options with their default values and allowed ranges. Descriptions of these options can be found in Appendix A.

Option	Default Value	Allowed Values
Optimization Algorithm Options		
MAXEVALS	50	[3–∞]
MAXITER	5	[1–∞]
MAXTIME	36,000	[1–∞]
POPSIZE	10	[2–∞]
SAMPLESIZE	50	[2–∞]
SEARCHMETHOD	GA	GA, LHS ,RANDOM, BAYESIAN
Validation Type Options		
FRACTION	0.3	[0.01–0.99]
KFOLD	5	[2–∞]
Objective Type Options		
OBJECTIVE	MSE (interval target) MISC (nominal target)	MSE, ASE, RASE, MAE, RMAE, MSLE, RMSLE (interval target) MISC, ASE, RASE, MCE, MCLL, AUC, F1, F05, GINI, GAMMA, TAU (nominal target)
TARGETEVENT	First event found	
Tuning Parameters Options		
USEPARAMETERS	COMBINED	COMBINED, STANDARD, CUSTOM
TUNINGPARAMETERS	N/A	
Other Options		
EVALHISTORY	TABLE	TABLE, LOG, NONE, ALL
NPARALLEL	0	[0–∞]

Table 2. Autotuning Options

The following example shows how you can use the AUTOTUNE statement to specify several custom definitions of hyperparameters to be tuned. You can change the initial value and the range of any tuning parameter, or you can prescribe a list of specific values to be used by the autotuning process.

```
proc gradboost data=mycaslib.dmagecr outmodel=mycaslib.mymodel;
    target good_bad / level=nominal;
    input checking duration history amount savings employed installp
        marital coapp resident property age other housing existcr job
        depends telephon foreign / level=interval;
    input purpose / level=nominal;
    autotune popsize=5 maxiter=3 objective=ASE
        tuningparameters=(
            ntrees(lb=10 ub=50 init=10)
            vars_to_try(values=4 8 12 16 20 init=4)
        );
run;
```

In general, the syntax for specifying custom definitions of hyperparameters to tune is

TUNINGPARAMETERS=*(<suboption> <suboption> …)*

where each <*suboption*> is specified as:

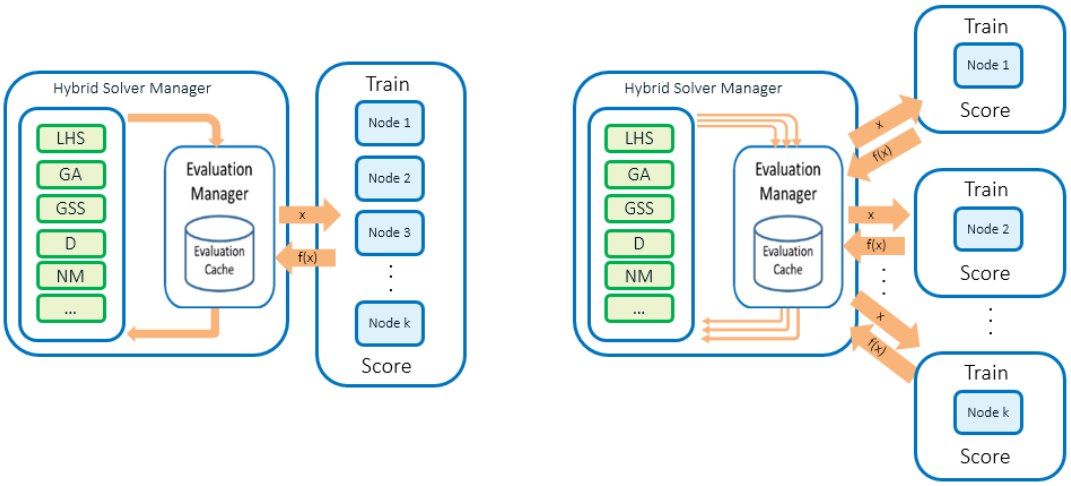
<*hyperparameter name*> (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)

Descriptions of these options can be found in Appendix A.

PARALLEL EXECUTION ON THE SAS VIYA PLATFORM

Hyperparameter tuning is ideally suited for the SAS Viya distributed analytics platform. The training of a model by a machine learning algorithm can be computationally expensive. As the size of a training data set grows, not only does the expense increase, but the data (and thus the training process) must often be distributed among compute nodes because they exceed the capacity of a single computer. Also, the configurations to be considered during tuning are independent, making a sequential tuning process not only expensive but unnecessary, assuming you have an available grid of compute resources. If a cross-validation process is chosen for model validation during tuning (which is typically necessary for small data sets), the tuning process cost is multiplied by a factor of *k* (the number of approximately equal-sized subsets, called folds), making a sequential tuning process even more intractable and reducing the number of configurations that can be considered.

Not only are the algorithms in SAS Visual Data Mining and Machine Learning designed for distributed analysis, but the local search optimization framework is also designed to take advantage of the distributed analytics platform, allowing distributed and concurrent training and scoring of candidate model configurations. When it comes to distributed/parallel processing for hyperparameter tuning, the literature typically presents two separate modes: “data parallel” (distributed/parallel training) and “model parallel” (parallel tuning). Truly big data requires distribution of the data and the training process. The diagram in Figure 8(a) illustrates this process: multiple worker nodes are used for training and scoring each alternative model configuration, but the tuning process is a sequential loop, which might also include another inner sequential loop for the cross-validation case. Because larger data sets are more expensive to train and score, even with a distributed data and training/scoring process, this sequential tuning process can be very expensive and restrictive in the number of alternatives that can feasibly be considered in a particular period of time. The “model parallel” case is shown in Figure 8(b): multiple alternative configurations are generated and evaluated in parallel, each on a single worker node, significantly reducing the tuning time. However, the data must fit on a single worker node.

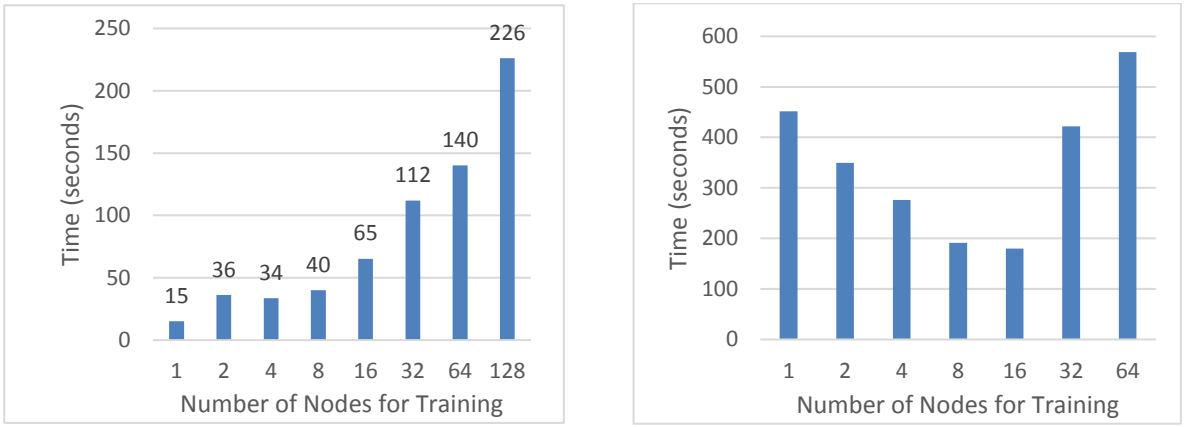


(a) "Data Parallel" (Sequential Tuning) **(b) "Model Parallel"**

Figure 8. Different Uses of Distributed Computing Resources

The challenge is to determine the best usage of available worker nodes. Ideally the best usage is a combination of the "data parallel" and "model parallel" modes, finding a balance of benefit from each. Example usage of a cluster of worker nodes for model tuning presents behaviors that can guide determination of the right balance. With small problems, using multiple worker nodes for training and scoring can actually reduce performance, as shown in Figure 9(a), where a forest model is tuned for the popular **iris** data set (150 observations) for a series of different configurations. The communication cost required to coordinate distributed data and training results continually increases the tuning time—from 15 seconds on a single machine to nearly four minutes on 128 nodes. Obviously this tuning process would benefit more from parallel tuning than from distributed/parallel training.

For large data sets, benefit is observed from distributing the training process. However, the benefit of distribution and parallel processing does not continue to increase with an increasing number of worker nodes. At some point the cost of communication again outweighs the benefit of parallel processing for model training. Figure 9(b) shows that for a credit data set of 70,000 observations, the time for training and tuning increases beyond 16 nodes, to a point where 64 nodes is more costly than 1 worker node.

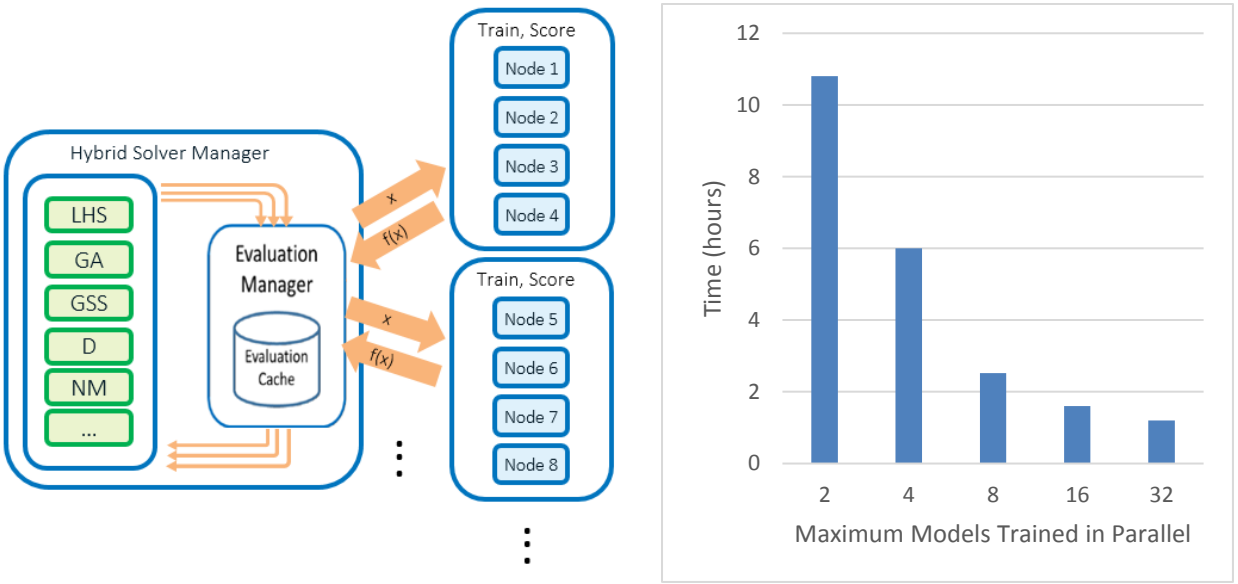


(a) Iris data set (105 / 45) **(b) Credit data set (49,000 / 21,000)**

Figure 9. Distributed Training with Sequential Tuning for Different Size Data Sets (Training/Validation)

When it comes to model tuning, the “model parallel” mode (training different model configurations in parallel) typically leads to larger gains in performance, especially with small- to medium-sized data sets. The performance gain is nearly linear as the number of nodes increases because each trained model is independent during tuning—no communication is required between the different configurations being trained. The number of nodes that are used is limited based on the size of the compute grid and the search strategy (for example, the population size at each iteration of a genetic algorithm). However, it is also possible to use both “data parallel” and “model parallel” modes through careful management of the data, the training process, and the tuning process. Because managing all aspects of this process in a distributed/parallel environment is very complex, using both modes is typically not discussed in the literature or implemented in practice. However, it is implemented in the SAS Visual Data Mining and Machine Learning autotune process.

As illustrated in Figure 10(a), multiple alternate model configurations are submitted concurrently by the local search optimization framework running on the SAS Viya platform, and the individual model configurations are trained and scored on a subset of available worker nodes so that multiple nodes can be used to manage large training data and speed up the training process. Figure 10(b) shows the time reduction for tuning when this process is implemented and the number of parallel configurations is increased, with each configuration being trained on four worker nodes. The tuning time for a neural network model that is tuned to handwritten data is reduced from 11 hours to just over 1 hour when the number of parallel configurations being tuned is increased from 2 (which uses 8 worker nodes) to 32 (which uses 128 worker nodes).



(a) Conceptual Framework **(b) Timing Results (70k observations)**

Figure 10. Distributed/Parallel Training and Parallel Tuning Combined

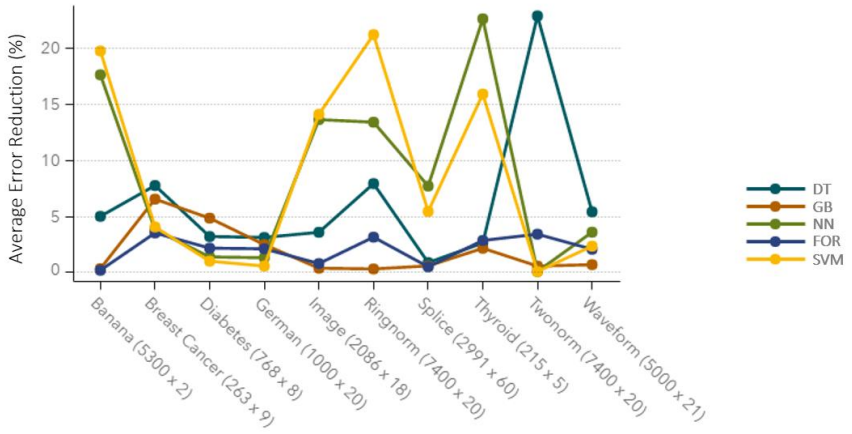
AUTOTUNING RESULTS AND RECOMMENDATIONS

This section presents tuning results for a set of benchmark problems, showing that the tuner is behaving as expected—model error is reduced when compared to using default hyperparameter values. This section also shows tuning time results for the benchmark problems and compares validation by single partition of the data to cross-validation. Finally, a common use case is presented—the tuning of a model to recognize handwritten digits. Code samples that demonstrate the application of autotuning to these and other problems can be found at https://github.com/sassoftware/sas-viya-machine-learning/autotuning.

BENCHMARK RESULTS

Figure 11 shows model improvement (error reduction or accuracy increase—higher is better) for a suite of 10 common machine learning test problems.[1] For this benchmark study, all problems are tuned with a 30% single partition for error validation during tuning, and the conservative default autotuning process is used: five iterations with only 10 configurations per iteration in LHS and GA. All problems are run 10 times, and the results that are obtained with different validation partitions are averaged in order to better assess behavior.

Here all problems are binary classification, allowing tuning of decision trees (DT), forests (FOR), gradient boosting trees (GB), neural networks (NN), and support vector machines (SVM). Figure 11 indicates that the tuner is working—with an average reduction in model error of 2% to more than 8% across all data sets, depending on model type, when compared to a baseline model that is trained with default settings of each machine learning algorithm. You can also see a hint of the "no free lunch" theorem (Wolpert 1996) with respect to different machine learning models for different data sets; no one modeling algorithm produces the largest improvement for all problems. Some modeling algorithms show 15–20% benefit through tuning. However, note that the baseline is not shown here, only the improvement. The starting point (the initial model error) is different in each case. The largest improvement might not lead to the lowest final model error. The first problem, the **Banana** data set, suggests that NN and SVM produce the largest improvement. The **Thyroid** problem shows a very wide range of improvement for different modeling algorithms.



Model Type	Average Improvement (%)
NN	8.53
SVM	8.45
DT	6.25
FOR	2.09
GB	1.91

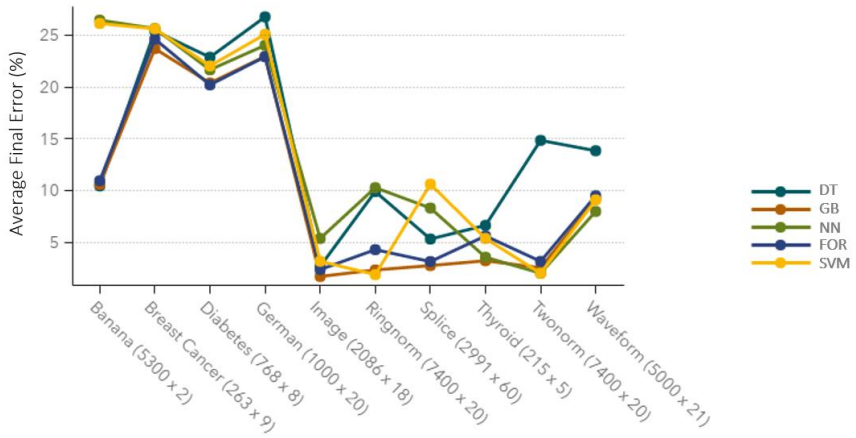
Figure 11. Benchmark Results: Average Improvement (Error Reduction) after Tuning

Figure 12 shows the final tuned model error—as averaged across the 10 tuning runs that use different validation partitions—for each problem and each modeling algorithm. The effect of the "no free lunch" theorem is quite evident here—different modeling algorithms are best for different problems. Consider the two data sets that were selected previously. For the **Banana** data set, you can see that although the improvement was best for NN and SVM, the final errors are highest for these two algorithms, indicating that the default models were worse for these modeling algorithms for this particular data set. All other modeling algorithms produce very similar error of around 10%—less than half the error from NN and SVM in this case. For the **Thyroid** data (which showed an even larger range of improvement for all modeling algorithms), the resulting model error is actually similar for different algorithms; again the default starting point is different, confirming the challenge of setting good defaults.

Overall, the benchmark results, when averaged across all data sets, are as expected. Decision trees are the simplest models and result in the highest overall average model error. If you build a forest of trees (a

[1] Data sets from http://mldata.org/repository/tags/data/IDA_Benchmark_Repository/, made available under the Public Domain Dedication and License v1.0, whose full text can be found at http://www.opendatacommons.org/licenses/pddl/1.0/ .

form of an ensemble model), you can reduce the error further, and for these data sets, the more complex gradient boosting training process leads to the lowest model error. The average errors for NN and SVM fall between the simple single decision tree and tree ensembles. Kernels other than linear or polynomial might be needed with SVM for these data sets, and neural networks might require more internal iterations or evaluation of more configurations, given the discrete combinations of hidden layers and units. *So why not always use gradient boosting?* Aside from fact that it might not be best for all data sets and the desire to use the simplest model that yields good predictions, there is a trade-off between resulting model accuracy and tuning time.

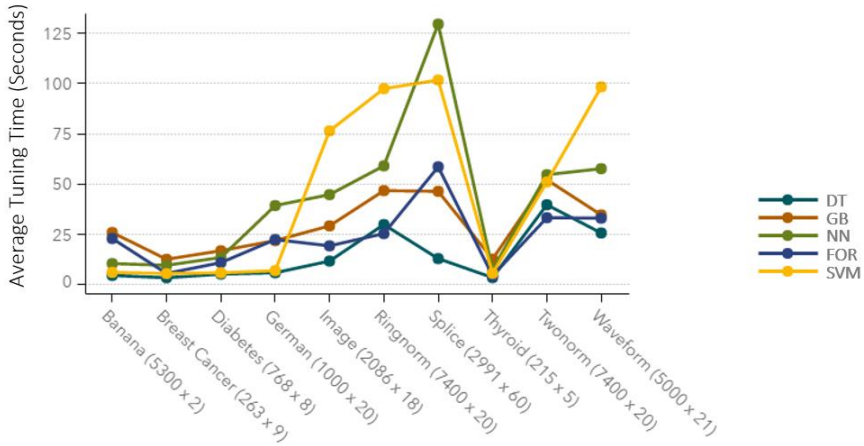


Model Type	Average Error After Tuning (%)
GB	9.9
FOR	10.7
SVM	13.1
NN	13.5
DT	13.9

Figure 12. Benchmark Results: Average Error after Tuning

TUNING TIME

For the tree-based algorithms, the trade-off is exactly the inverse ranking of machine learning algorithms with time compared to accuracy on average, as shown in Figure 13. Decision trees are the simplest and most efficient—only 14.4 seconds here for full tuning with this conservative tuning process. Building a forest of trees increases the time to over 23 seconds, and the complex gradient boosting process is more expensive at 30 seconds average tuning time. NN and SVM tuning times are similar for several problems, but higher for some, leading to a higher overall average tuning time; both use iterative optimization schemes internally to train models, and convergence might take longer for some data sets.



Model Type	Average Tuning Time (Seconds)	Average Parallel Speed-Up
DT	14.4	3.6
FOR	23.7	5.1
GB	30.0	4.7
NN	42.7	4.1
SVM	45.6	4.6

Figure 13. Benchmark Results: Average Total Tuning Time in Seconds

For these benchmark data sets, the tuning time is manageable—less than 30 seconds for fully tuning most models. Even the worst case, a neural network tuned to the wide **Splice** data set (which has 60 attributes) is tuned in just over two minutes. Note here again that all configurations are trained in parallel during each iteration of tuning. The total CPU time for this worst-case tuning is closer to eight minutes. With the default tuning process of 10 configurations during each of five iterations, one configuration is carried forward each iteration; so up to nine new configurations are evaluated in parallel at each iteration (by default). Figure 13 also shows parallel speed-up time (which is the total CPU time divided by the tuner clock time) of 3X–5X speed-up with parallel tuning. *Why is the speed-up not 9X with nine parallel evaluations?* Putting aside some overhead of managing parallel model training, the longest running configuration of the nine models that are trained in parallel determines the iteration time. For example, if eight configurations take 1 second each for training, and the ninth takes 2 seconds, a sequential training time of 10 seconds is reduced to 2 seconds, the longest-running model training. A 5X speed-up is observed rather than the average of the nine training times (1.1 seconds), which would be a 9X speed-up.

For larger data sets, longer-running training times, and an increased number of configurations at each iteration, the parallel speed-up will increase. For these benchmark problems, running in parallel on a compute grid might not be necessary; for a 30-second tuning time, 5X longer sequentially might not be a concern. Eight minutes for tuning the longer-running data sets might not even be a concern. Before you consider parallel/distributed training and tuning for larger data sets, however, you need to consider another tuning cost with respect to the validation process: cross-validation.

CROSS-VALIDATION

For small data sets, a single validation partition might leave insufficient data for validation in addition to training. Keeping the training and validation data representative can be a challenge. For this reason, cross-validation is typically recommended for model validation. With cross-validation, the data are partitioned into *k* approximately equal subsets called *folds*; training/scoring happens *k* times—training on all except the current holdout fold, and scoring on the holdout fold. The cross-validation error is then an average of the errors obtained from each validation fold.

This process can produce a better representation of error across the entire data set, because all observations are used for training and scoring. Figure 14 shows a comparison of cross-validation errors and the errors from a single partition, where both are compared to errors from a separate test set. The three smallest data sets are chosen, and the value in parentheses indicates the size of the holdout test set. Gradient boosting tree models are tuned in this case. The plot shows the absolute value of the error difference, where lower is better (validation error closer to test error). For the **Breast Cancer** data set, the single partition results and the cross-validation results are nearly equal. However, for the other two data sets, the cross-validation process that uses five folds produces a better representation of test error than the single validation partition does—in both cases, the cross-validation error is more than 5% closer to the test error.

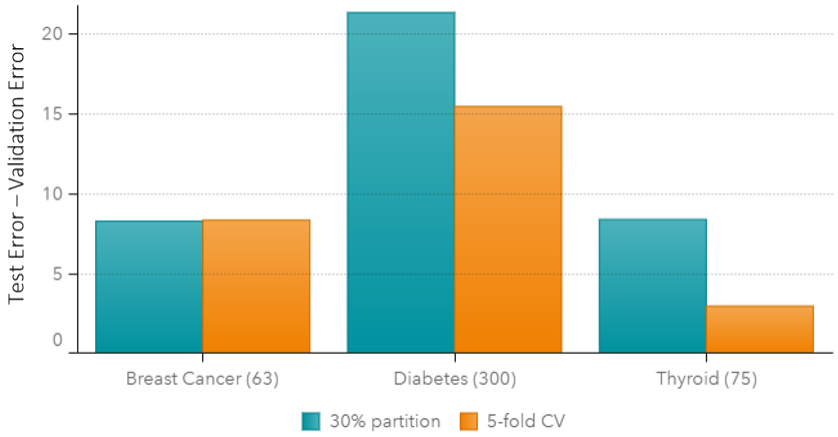


Figure 14. Benchmark Results: Single Partition versus Cross-Validation

With this cross-validation process, the trade-off is again increased time. The model training time, and therefore the overall tuning time, is increased by a factor of *k*. Thus, a 5X increase in time with sequential tuning for a small data set and a 5X increase with five-fold cross-validation becomes a 25X increase in tuning time. So tuning a model to even a small data set can benefit from parallel tuning.

TUNING MODELS FOR THE MNIST DIGITS DATA

In this section, the power of combined distributed modeling training and parallel tuning enabled by the SAS Viya distributed analytics platform is demonstrated by using the popular MNIST (Mixed National Institute of Standards and Technologies) database of handwritten digits (Lecun, Cortes, and Burges 2016). This database contains digitized representations of handwritten digits 0–9, in the form of a 28 × 28 image for a total of 784 pixels. Each digit image is an observation (row) in the data set, with a column for each pixel containing a grayscale value for that pixel. The database includes 60,000 observations for training, and a test set of 10,000 observations. Like many studies that use this data set, this example uses the test set for model validation during tuning.

The GRADBOOST procedure is applied to the digits database with autotuning according to the configuration that is specified in the following statements:

```
proc gradboost data=mycaslib.digits;
    partition rolevar=validvar(train='0' valid='1');
    input &inputnames;
    target label / level=nominal;
    autotune popsize=129 maxiter=20 maxevals=2560
             nparallel=32 maxtime=172800
             tuningparameters=(ntrees(ub=200));
run;
```

In this example, the training and test data sets have been combined, with the ROLEVAR= option specifying the variable that indicates which observations to use during training and which to use during scoring for validation. The PARTITION statement is used in conjunction with the AUTOTUNE statement to specify the validation approach—a single partition in this case, but using the ROLEVAR= option instead of a randomly selected percentage validation fraction. Because there are 784 potential inputs (pixels) and some of the pixels are blank for all observations, the list of input pixels that are not blank is preprocessed into the macro variable *&inputnames*, resulting in 719 inputs (see the code in Appendix B). For tuning, the number of configurations to try has been significantly increased from the default settings. Up to 20 iterations are requested, with a population size (number of configurations per iteration) of 129. Recall that one configuration is carried forward each iteration, so this specification results in up to 128 new configurations evaluated in each iteration.

A grid with 142 nodes is employed and configured to use four worker nodes per model training. *Why four instead of eight or 16 worker nodes per training as suggested in Figure 9?* There is a trade-off here for node assignment: training time versus tuning time. Using four worker nodes per training and tuning 32 models in parallel uses 128 worker nodes in total. If the number of worker nodes for training is doubled, the number of parallel models might need to be reduced in order to balance the load. Here it is decided that the gain from doubling the parallel tuning is larger than the reduced training time from doubling the number of worker nodes for each model training. Using four worker nodes, the training time for a default gradient boosting model is approximately 21.5 minutes. With eight worker nodes, the training time is approximately 13 minutes.

With up to 20 iterations and 128 configurations per iteration, the MAXEVALS= option is increased to 2,560 to accommodate these settings (the default for this option is 50, which would lead to termination before the first iteration finishes). The MAXTIME= option is also increased to support up to 48 hours of tuning time; many of the configurations train in less than the time required for the default model training.

Finally, the upper bound on the tuning range for the NTREES hyperparameter is increased to 200 from the default value of 150. The syntax enables you to override either or both of the hyperparameter bounds; in this example, the default lower bound for NTREES is unchanged and PROC GRADBOOST uses default settings for the other five tuning parameters. Increasing the upper bound for the *number of trees* hyperparameter will increase the training time for some models (and thus increase the tuning time) but might allow better models to be identified.

Some of the challenges of hyperparameter tuning discussed earlier can be seen in Figure 15, which shows the error for the configurations that are evaluated in the first iteration of tuning. Recall that the first iteration uses a Latin hypercube sample (which is more uniform than a pure random sample) to obtain an initial sample of the space. Two key points can be seen very clearly in this plot:

- The majority of the evaluated configurations produce a validation error larger than that of the default configuration, which is 2.57%.
- As you look across the plot, you can clearly see that many different configurations produce very similar error rates. These similar error rates indicate flat regions in the space, which are difficult for an optimizer to traverse and make it difficult for random configurations to identify an improved model.

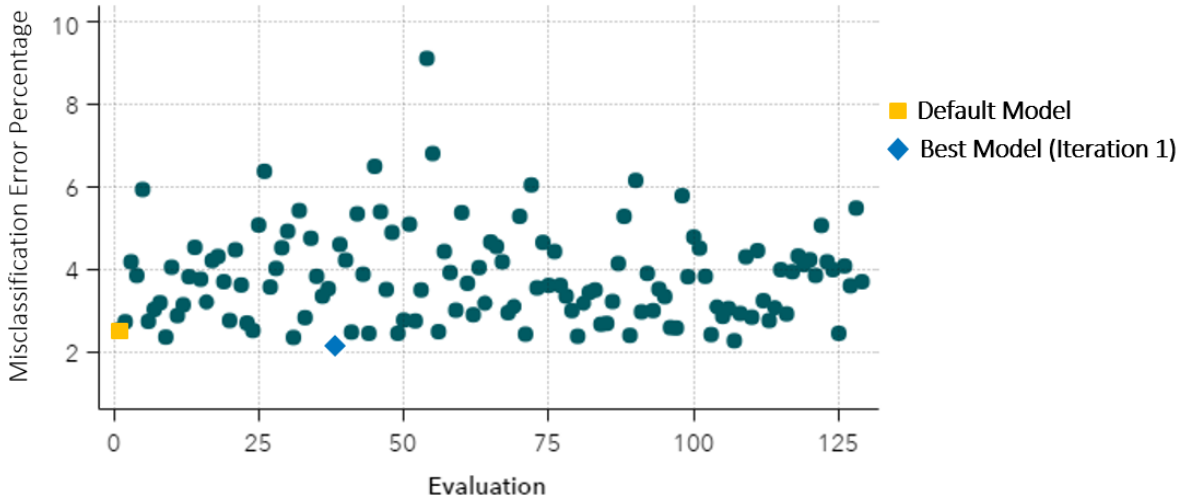


Figure 15. The GRADBOOST Procedure Tuning to MNIST Digits Data—Iteration 1

An improved model is found in the first iteration, with an error of 2.21%. Figure 16 shows the results of applying the genetic algorithm in subsequent iterations. The error is reduced again in 11 of the remaining 19 iterations. The tuning process is terminated when the maximum requested number of iterations is reached, after evaluating 2,555 unique model configurations. Here the final error is 1.74%. Details of the final model configuration are shown in Figure 17. The *number of trees* hyperparameter (which starts with a default of 100 trees) is driven up to 142 trees, still below the default upper bound of 150. Only 317 variables are used, well below the default of all (719) variables. *Learning rate* is increased from a default of 0.1 to 0.19, and *sampling rate* is increased from 0.5 to 1.0, its upper bound. Both lasso and ridge regularization begin at 0; *lasso* is increased to 0.14 and *ridge* is increased to 0.23.

Also shown in Figure 17 are tuning timing information and a tuning process summary. You can see that the tuning time of just over 28 hours (101,823 seconds) actually uses more than 760 hours of CPU time (the sum of all parallel training/scoring time for each objective evaluation), which results in a parallel speed-up of nearly 27X—much more than the 5X best case speed-up that is seen with the benchmark problems, and a much better ratio of 0.84 (with 32 parallel evaluations) compared to 0.56 (5X speed-up with 9 parallel evaluations).

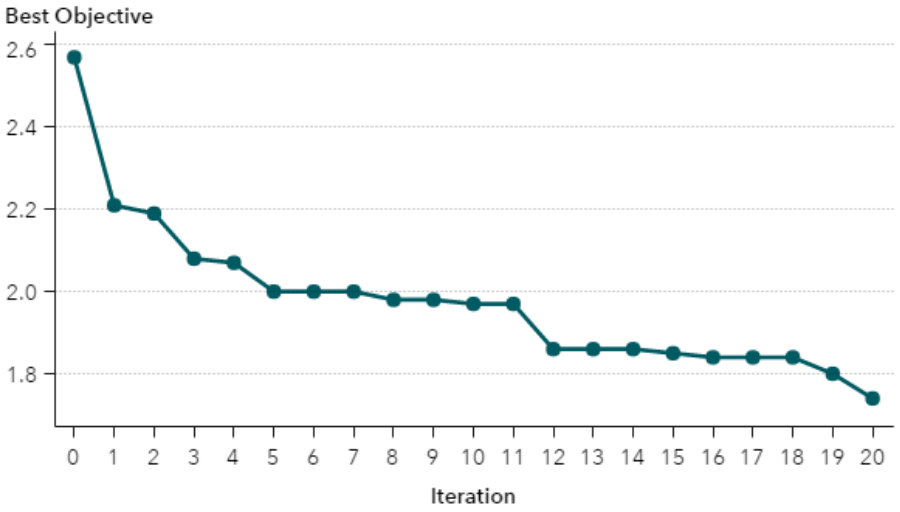


Figure 16. The GRADBOOST Procedure Tuning Iteration History, MNIST Digits Data

Best Configuration	
Evaluation	2551
Number of Trees	142
Number of Variables to Try	317
Learning Rate	0.19165378
Sampling Rate	1
Lasso	0.13883111
Ridge	0.2295815
Misclassification Error Percentage	1.74

Tuner Task Timing		
Task	Seconds	Percent
Model Training	2709131	98.98
Model Scoring	28018.86	1.02
Total Objective Evaluations	2737150	100.00
Tuner	26.84	0.00
Total CPU Time	2737177	100.00

Tuner Summary	
Initial Configuration Objective Value	2.5700
Best Configuration Objective Value	1.7400
Worst Configuration Objective Value	24.7200
Initial Configuration Evaluation Time in Seconds	1292.92
Best Configuration Evaluation Time in Seconds	1087.17
Number of Improved Configurations	18
Number of Evaluated Configurations	2555
Total Tuning Time in Seconds	101823
Parallel Tuning Speedup	26.8816

Figure 17. The GRADBOOST Procedure Tuning Results, MNIST Digits Data

CONCLUSION

The explosion of digital data is generating many opportunities for big data analytics, which in turn provides many opportunities for tuning predictive models to capitalize on the information contained in the data—to make better predictions that lead to better decisions. The tuning process often leads to hyperparameter settings that are better than the default values. But even when the default settings do work well, the hyperparameter tuning process provides a heuristic validation of these settings, giving you greater assurance that you have not overlooked a model configuration that has higher accuracy. This validation is of significant value itself.

The SAS Viya distributed analytics platform is ideally suited for tuning predictive models because many configurations often need to be evaluated. The TREESPLIT, FOREST, GRADBOOST, NNET, SVMACHINE, and FACTMAC procedures implement a fully automated tuning process that requires only the AUTOTUNE keyword to perform a conservative tuning process. This implementation includes the most commonly tuned parameters for each machine learning algorithm. You can adjust the ranges or list of values to try for these hyperparameters, exclude hyperparameters from the tuning process, and configure the tuning process itself. The local search optimization framework that is used for tuning is also ideally suited for use on the SAS Viya platform; alternate search methods can be applied and combined, with the framework managing concurrent execution and information sharing. With the complexity of the

model-fitting space, many search strategies are under investigation for both effective and efficient identification of good hyperparameter values. Bayesian optimization is currently popular for hyperparameter optimization, and an experimental algorithm is available in the local search optimization framework. However, the key feature of local search optimization is its ability to build hybrid strategies that combine the strengths of *multiple* methods; no one search method will be best for tuning for all data sets and all machine learning algorithms—there is “no free lunch.”

The distributed execution capability provided by the SAS Viya platform is fully exploited in this autotuning implementation. With small data sets that might not require distributed training, the need for and added expense of cross-validation support the use of parallel tuning to balance the added expense. For large data sets, distributed/parallel training and parallel model tuning can be applied concurrently within the platform for maximum benefit. One challenge is selecting the right combination of the number of worker nodes per model training and the number of parallel model configurations. With small data sets, the number of workers per training should be set as low as possible and the number of parallel configurations as high as possible, allowing the compute grid nodes to be used for parallel *tuning*. With larger data sets, such as the MNIST digits data set, a balance must be struck. Usually hundreds of worker nodes are not needed for a single model training (even with truly big data) and there is always a communication cost that can be detrimental if too many nodes are used for training. With the number of configurations evaluated in parallel, there are never “too many”—the more configurations that are evaluated in parallel, the closer to 100% efficiency the tuning process becomes, *given that many parallel configurations are not all evaluated on the same worker nodes* (evaluating hundreds of configurations on four worker nodes simultaneously will slow the process down). Approximately 84% efficiency was achieved when the PROC GRADBOOST tuning process was used to model the MNIST digits data set.

What is not discussed and demonstrated in this paper is a comparison of the implemented hybrid strategy with a random search approach for hyperparameter tuning. Random search is popular for two main reasons: a) the hyperparameter space is often discrete, which does not affect random search, and b) random search is simple and all configurations could potentially be evaluated concurrently because they are all independent. The latter reason is a strong argument when a limited number of configurations is considered or a very large grid is available. In the case of the GRADBOOST procedure tuning a model to the MNIST digits data, four nodes per training and 32 parallel configurations uses 128 nodes. The best solution was identified at evaluation 2,551. These evaluations could not have all been performed in parallel. With a combination of discrete and continuous hyperparameters, the hybrid strategy that uses a combination of Latin hypercube sampling (LHS) and a genetic algorithm (GA) is powerful; this strategy exploits the benefits of a uniform search of the space and evolves the search using knowledge gained from previous configurations. The local search optimization framework also supports random, LHS, and Bayesian search methods.

With an ever-growing collection of powerful machine learning algorithms, all governed by hyperparameters that drive their fitness quality, the “no free lunch” theorem presents yet another challenge: deciding which machine learning algorithm to tune to a particular data set. This choice is an added layer of tuning and model selection that could be managed in a model tuning framework, with parallel tuning across multiple modeling algorithms in addition to multiple configurations. Combining models of different types adds a dimension of complexity to explore with tuning. With so many variations to consider in this process, careful management of the computation process is required.

APPENDIX A: DESCRIPTION OF AUTOTUNE STATEMENT OPTIONS

You can specify the following options in the AUTOTUNE statement:

MAXEVALS=*number* specifies the maximum number of configuration evaluations allowed for the tuner.

MAXITER=*number* specifies the maximum number of iterations of the optimization tuner.

MAXTIME=*number* specifies the maximum time (in seconds) allowed for the tuner.

POPSIZE=*number* specifies the maximum number of configurations to evaluate in one iteration (population).

SAMPLESIZE=*number* specifies the total number of configurations to evaluate when SEARCHMETHOD=RANDOM or SEARCHMETHOD=LHS.

SEARCHMETHOD=*search-method-name* specifies the search method to be used by the tuner.

FRACTION=*number* specifies the fraction of all data to be used for validation.

KFOLD=*number* specifies the number of partition folds in the cross-validation process.

EVALHISTORY=*eval-history-option* specifies the location in which to report the complete evaluation (the ODS table only, the log only, both places, or not at all).

NPARALLEL=*number* specifies the number of configurations to be evaluated by the tuner simultaneously.

OBJECTIVE=*objective-option-name* specifies the measure of model error to be used by the tuner when it searches for the best configuration.

TARGETEVENT=*target-event-name* specifies the target event to be used by the ASSESS algorithm when it calculates the error metric (used only for nominal target parameters).

USEPARAMETERS=*use-parameter-option* specifies the set of parameters to tune, with *use-parameter-option* specified as:

STANDARD tunes using the default bounds and initial values for all parameters.

CUSTOM tunes only the parameters that are specified in the TUNINGPARAMETERS= option.

COMBINED tunes the parameters that are specified in the TUNINGPARAMETERS= option and uses default bounds and initial values to tune all other parameters.

TUNINGPARAMETERS=*(suboption . . . <suboption>)* specifies the hyperparameters to tune and which ranges to tune over, with *suboption* specified as:

NAME (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE), where

LB specifies a custom lower bound to override the default lower bound.

UB specifies a custom upper bound to override the default upper bound.

VALUES specifies a list of values to try for this hyperparameter

INIT specifies the value to use for training a baseline model.

EXCLUDE specifies that this hyperparameter should *not* be tuned; it will remain fixed at the value specified for the procedure (or default if none is specified).

APPENDIX B: CODE TO CREATE A LIST OF NONEMPTY PIXELS FOR MNIST DIGITS

```
proc cardinality data=mycas.digits outcard=mycas.digitscard;
run;

proc sql;
  select _varname_ into :inputnames separated by ' '
    from mycas.digitscard
    where _mean_ > 0
      and _varname_ contains "pixel"
    ;
quit;
```

REFERENCES

Bergstra, J., and Bengio, Y. (2012). “Random Search for Hyper-parameter Optimization.” *Journal of Machine Learning Research* 13:281–305.

Bottou, L., Curtis, F. E., and Nocedal, J. (2016). “Optimization Methods for Large-Scale Machine Learning.” arXiv:1606.04838 [stat.ML].

Dewancker, I., McCourt, M., Clark, S., Hayes, P., Johnson, A., and Ke, G. (2016). “A Stratified Analysis of Bayesian Optimization Methods.” arXiv:1603.09441v1 [cs.LG].

Gomes, T. A. F., Prudêncio, R. B. C., Soares, C., Rossi, A. L. D., and Carvalho, A. (2012) “Combining Meta-learning and Search Techniques to Select Parameters for Support Vector Machines.” *Neurocomputing* 75:3–13.

Konen, W., Koch, P., Flasch, O., Bartz-Beielstein, T., Friese, M., and Naujoks, B. (2011). “Tuned Data Mining: A Benchmark Study on Different Tuners.” In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (GECCO-2011). New York: SIGEVO/ACM.

LeCun, Y., Cortes, C., and Burges, C. J. C. (2016). “The MNIST Database of Handwritten Digits.” Accessed April 8, 2016. http://yann.lecun.com/exdb/mnist/.

Lorena, A. C., and de Carvalho, A. C. P. L. F. (2008). “Evolutionary Tuning of SVM Parameter Values in Multiclass Problems.” *Neurocomputing* 71:3326–3334.

McKay, M. D. (1992). “Latin Hypercube Sampling as a Tool in Uncertainty Analysis of Computer Models.” In *Proceedings of the 24th Conference on Winter Simulation* (WSC 1992), edited by J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, 557–564. New York: ACM.

Renukadevi, N. T., and Thangaraj, P. (2014). “Performance Analysis of Optimization Techniques for Medical Image Retrieval.” *Journal of Theoretical and Applied Information Technology* 59:390–399.

Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P. (1989). “Design and Analysis of Computer Experiments.” *Statistical Science* 4:409–423.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. E. (2013). “On the Importance of Initialization and Momentum in Deep Learning.” In *Proceedings of the 30th International Conference on Machine Learning* (ICML-13), edited by S. Dasgupta and D. McAllester, 1139–1147. International Machine Learning Society.

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.

Wolpert, D. H. (1996). “The Lack of A Priori Distinctions between Learning Algorithms.” *Neural Computation* 8:1341–1390.

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. https://support.sas.com/resources/papers/proceedings16/SAS2360-2016.pdf.

ACKNOWLEDGMENTS

The authors would like to thank Joshua Griffin, Scott Pope, and Anne Baxter for their contributions to this paper.

RECOMMENDED READING

- *Getting Started with SAS Visual Data Mining and Machine Learning 8.1*
- *SAS Visual Data Mining and Machine Learning 8.1: Data Mining and Machine Learning Procedures*
- *SAS Visual Data Mining and Machine Learning 8.1: Statistical Procedures*

CONTACT INFORMATION

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

Patrick Koch	Brett Wujek	Oleg Golovidov	Steven Gardner
SAS Institute Inc.	SAS Institute Inc.	SAS Institute Inc.	SAS Institute Inc.
patrick.koch@sas.com	brett.wujek@sas.com	oleg.golovidov@sas.com	steven.gardner@sas.com