Reading: Reference guide: XGBoost tuning

Previously, you learned about gradient boosting machine models and studied how to build and tune them with XGBoost's scikit-learn API. This reading is a quick-reference guide to help you when you're building XGBoost models of your own. It includes information on the following components:

- Import statements
- Hyperparameters

Import statements

The following are some of the most commonly used import statements for gradient boosting models using the XGBoost library together with scikit-learn.

Models

For classification tasks:

```
from xgboost import XGBClassifier
```

For regression tasks:

```
from xgboost import XGBRegressor
```

Evaluation metrics

For classification tasks:

```
from sklearn.metrics import
```

accuracy_score(y_true, y_pred, *[, ...])	Accuracy classification score

average_precision_score(y_true, ...)	Compute average precision (AP) from prediction scores
confusion_matrix(y_true, y_pred, *)	Compute confusion matrix to evaluate the performance of the training of a model
f1_score(y_true, y_pred, *[, ...])	Compute the F1 score, also known as balanced F-score or F-measure
fbeta_score(y_true, y_pred, *, beta)	Compute the F-beta score
metrics.log_loss(y_true, y_pred, *[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss
multilabel_confusion_matrix(y_true, ...)	Compute a confusion matrix for each class or sample
precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds
precision_score(y_true, y_pred, *[, ...])	Compute the precision
recall_score(y_true, y_pred, *[, ...])	Compute the recall
roc_auc_score(y_true, y_score, *[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores

For regression tasks:

```
from sklearn.metrics import
```

`mean_absolute_error`(y_true, y_pred, *)	Mean absolute error regression loss
`mean_squared_error`(y_true, y_pred, *)	Mean squared error regression loss
`mean_squared_log_error`(y_true, y_pred, *)	Mean squared logarithmic error regression loss
`median_absolute_error`(y_true, y_pred, *)	Median absolute error regression loss
`mean_absolute_percentage_error`(...)	Mean absolute percentage error (MAPE) regression loss
`r2_score`(y_true, y_pred, *[, ...])	R^2 (coefficient of determination) regression score function

Hyperparameters

The following are some of the most important hyperparameters for gradient boosting machine classification models built with the XGBoost library. These are the hyperparameters that data professionals typically reach for first, because they are among the most intuitive and they control the model at different levels using a diverse variety of mechanisms.

`n_estimators`

Hyperparameter	What it does	Input type	Default Value
`n_estimators`	Specifies the number of boosting rounds (i.e., the number of trees your model will build in its ensemble)	int	100

Considerations:

A typical range is 50–500. Consider how much data you have, how deep the trees are allowed to grow, and how many samples are bootstrapped from the overall data to grow each tree (you generally need more trees if they're shallow, and more trees if your bootstrap sample size represents just a small fraction of your overall data). For an extreme but illustrative example, if you have a dataset of 10,000, and each tree only bootstraps 20 samples, you'll need more trees than if you gave each tree 5,000 samples. Also keep in mind that, unlike random forest, which can grow base learners in parallel, gradient boosting grows base learners successively, so training can take longer for more trees.

`max_depth`

Hyperparameter	What it does	Input type	Default Value
`max_depth`	Specifies how many levels your base learner trees can have. If None, trees grow until leaves are pure or until all leaves have less than `min_child_weight`.	int	3

Considerations: Controls complexity of the model. Gradient boosting typically uses weak learners, or "decision stumps" (i.e., shallow trees). Restricting tree depth can reduce training times and serving latency as well as prevent overfitting. Consider values 2–6.

`min_child_weight`

Hyperparameter	What it does	Input type	Default Value
`min_child_weight`	Controls threshold below which a node becomes a leaf, based on the combined weight of the samples it contains. For regression models, this value is functionally equivalent to a number of samples. For the binary classification objective, the	int or float	1

	weight of a sample in a node is dependent on its probability of response as calculated by that tree. The weight of the sample decreases the more certain the model is (i.e., the closer the probability of response is to 0 or 1).		

Considerations: Higher values will stop trees splitting further, and lower values will allow trees to continue to split further. If your model is underfitting, then you may want to lower it to allow for more complexity. Conversely, increase this value to stop your trees from getting too finely divided.

`learning_rate`

Hyperparameter	What it does	Input type	Default Value
`learning_rate`	Controls how much importance is given to each consecutive base learner in the ensemble's final prediction. Also known as *eta* or *shrinkage*.	float	0.1

Considerations: Values can range from (0–1]. Typical values range from 0.01 to 0.3. Lower values mean less weight is given to each consecutive base learner. Consider how many trees are in your ensemble. Lower values typically benefit from more trees.

`colsample_bytree*`

Hyperparameter	What it does	Input type	Default Value
`colsample_bytree*`	Specifies the percentage (0–1.0] of features that each tree randomly selects during training	float	1.0

Considerations: Adds randomness to the model to make it robust to noise. Consider how many features the dataset has and how many trees will be grown. Fewer features sampled

means more base learners might be needed. Small colsample_bytree values on datasets with many features mean more unpredictive trees in the ensemble.

`subsample*`

Hyperparameter	What it does	Input type	Default Value
`subsample*`	Specifies the percentage (0–1.0] of observations sampled from the dataset to train each base model.	float	1.0

Considerations: Adds randomness to the model to make it robust to noise. Consider the size of your dataset. When working with large datasets, it can be beneficial to limit the number of samples in each tree, because doing so can greatly reduce training time and yet still result in a robust model. For example, 20% of 1 billion might be enough to capture patterns in the data, but if you only have 1,000 samples in your dataset then you'll probably need to use them all.

*Note that `colsample_bytree` and `subsample` were not used in the Tune a GBM model video and its accompanying notebook; they are included here so you can use these hyperparameters in your own work. Remember that using fractions of the data to train each base learner can possibly improve model predictions and certainly speed up training times.

Key takeaways

When building machine learning models, it's essential to have the right tools and understand how to use them. Although there are numerous other hyperparameters to explore, the ones in this reference guide are among the most important. Be inquisitive and try different approaches. Discovering ways to improve your model is a lot of fun!

Resources for more information

More detailed information about XGBoost can be found here:

- scikit-learn model metrics: documentation for evaluation metrics in scikit-learn

- XGBoost classifier: XGBoost documentation for classification tasks using the scikit-learn API
- XGBoost Regressor: XGBoost documentation for regression tasks using the scikit-learn API
- Notes on parameter tuning from XGBoost
- XGBoost parameters: XGBoost parameters guide. **NOTE:** The information in this link is not specific to the scikit-learn API. **The default values listed in this resource are not always the same as the ones in the scikit-learn API.**