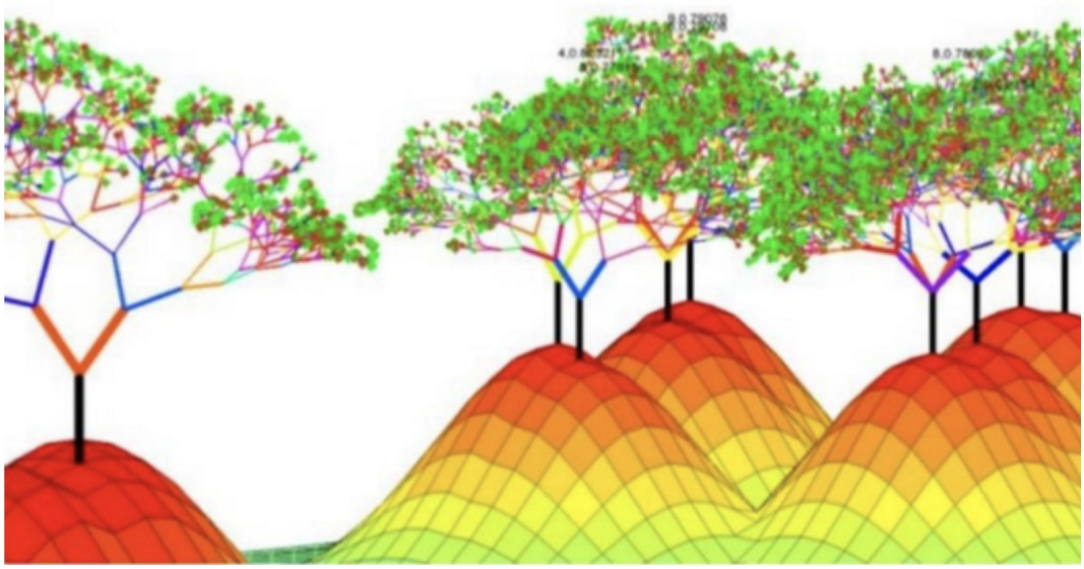
From Zero to Hero in XGBoost Tuning

A walk through some of the most common (and not so common too!) hyperparameters of XGBoost

Florencia
Leoni
May 21 · 7 min read ★



Boosted Trees by Chen Shikun.

XGBoost or Extreme Gradient Boosting is an optimized implementation of the Gradient Boosting algorithm. Since its introduction in 2014 XGBoost has been the darling of machine learning hackathons and competitions because of its prediction performance and processing time.

It all started with Boosting…

Boosting is a type of Ensemble technique. Ensemble Learning borrows from Condorcet's Jury Theorem the idea of the *wisdom of crowds*. Thus Ensemble techniques combine the results of different models to improve the overall results and performance

> *A majority vote classifies correctly with higher probability than any person (model), and as the number of people (models) becomes large, the accuracy of the majority vote approaches 100%—Scott Page, The Model Thinker.*

In decision-tree based machine learning, Boosting algorithms implement a sequential process where each model attempts to correct the mistakes of the previous models. The idea is to convert many *weak learners* into one *strong learner.* Gradient Boosting employs the gradient descent algorithm to minimize errors in sequential models. The major inefficiency in Gradient Boosting is that it creates one decision tree at a time.

To overcome this, Tianqi Chen and Carlos Guestrin built *A Scalable Tree Boosting System*—XGBoost can be thought of as Gradient Boosting on steroids. It features parallelized tree building, cache-aware access, sparsity awareness, regularization, and weighted quantile sketch as some of its systems optimization and algorithmic enhancements.

> *When in doubt, use XGBoost—Words of Wisdom by Owen Zhang.*

To walk you through XGBoost and its hyperparameters, we'll build a simple classification model using the Fashion MNIST dataset.

Preprocessing the data

The Fashion MNIST dataset consists of a training set of 60,000 28x28 greyscale images associated with 10 classification labels, and a test set of 10,000 images. The labels are the following:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

Let's take a look at the first and last image.

```
# View the first image
first_image = features_train.iloc[0]
first_image = np.array(first_image, dtype = "float")
pixels = first_image.reshape((28, 28))
plt.imshow(pixels, cmap = "gray_r")
plt.axis("off")
plt.show()
print ("Label:", target_train.iloc[0])

# View last image
```

```
last_image = features_train.iloc[-1]
```

Label: 2

Label: 7

Each instance (image) in the dataset has 784 features (one for each pixel) and the value in each feature ranges from 0 to 255, hence we'll use Scikit-Learn's `StandardScaler` to rescale the values to a smaller range with mean zero and unit variance.

```
# Use sklearn StandardScaler to scale pixel values
from sklearn.preprocessing import StandardScaler
# Create scale object
scaler = StandardScaler()
# Fit scaler to training data only
scaler_fit = scaler.fit(features_train)
# Transform both train and test data with the trained scaler
X_train = scaler_fit.transform(features_train)
X_test = scaler_fit.transform(features_test)
```

scale_images.py hosted with ❤ by **GitHub** **view raw**

Baseline Model

We are going to use an `XGBoostClassifier` to predict the label of a given image. Building an `XGBoostClassifier` is pretty straightforward. We'll start with a simple baseline model and move on from there.

For this first iteration, we'll only specify one hyperparameter: objective and set it to `"multi:softmax"`. The `objective` is part of the Learning Task hyperparameters, and it specifies the learning task (regression, classification, ranking, etc) and function to be used.

Let's backtrack for a second. What are hyperparameters?—They are the parameters that are initialized before training a model because they cannot be learned from the algorithm. They control the behavior of the training algorithm and have a high impact on the performance of a model. The typical metaphor goes like this: *hyperparameters are the knobs one turns to tweak a machine learning model*. They are essential to optimization and to improve evaluation metrics.

And now, let's take a look at our model and the results.

```
import xgboost as xgb
# Create XGB Classifier object
xgb_clf = xgb.XGBClassifier(objective = "multi:softmax")
# Fit model
xgb_model = xgb_clf.fit(X_train, target_train)
# Predictions
y_train_preds = xgb_model.predict(X_train)
y_test_preds = xgb_model.predict(X_test)
# Print F1 scores and Accuracy
print("Training F1 Micro Average: ", f1_score(target_train, y_train_preds, average = "micro
print("Test F1 Micro Average: ", f1_score(target_test, y_test_preds, average = "micro"))
```

```
Training F1 Micro Average:  0.8794833333333333
Test F1 Micro Average:  0.8674
Test Accuracy:  0.8674
```

Not too shabby! But we can try to beat these first scores with some tweaking and *knob-turning*.

Tuning like a Boss!

One of XGBoost's strongest advantages is the degree of customization available, *i.e.* an intimidatingly long list of hyperparameters you can tune, which were designed mostly to prevent overfitting.

The million dollar question, in this case, is what to tune and how? There are no benchmarks as to what the ideal hyperparameters are since these will depend on your specific problem, your data, and what you're optimizing for. But once you understand the concepts of even the most obscure hyperparameters you'll be well on your way to **tuning like a boss**!

To find our best hyperparameters we can use Scikit-Learn's `RandomizedSearchCV` or `GridSearchCV`. The difference between the two is a trade-off between lower run-time and better performance, respectively. Taking the size of the dataset into consideration `RandomizedSearchCV` is the way to go this time around.

```
from sklearn.model_selection import RandomizedSearchCV
import xgboost as xgb

# Create XGB Classifier object
xgb_clf = xgb.XGBClassifier(tree_method = "gpu_exact", predictor = "gpu_predictor", verbosi
                            eval_metric = ["merror", "map", "auc"], objective = "multi:softma
# Create parameter grid
parameters = {"learning_rate": [0.1, 0.01, 0.001],
              "gamma" : [0.01, 0.1, 0.3, 0.5, 1, 1.5, 2],
              "max_depth": [2, 4, 7, 10],
              "colsample_bytree": [0.3, 0.6, 0.8, 1.0],
```

We start by creating an `XGBClassifier` object, just as with the baseline model, we are passing the hyperparameters `tree_method`, `predictor`, `verbosity` and `eval_metric`, in addition to the `objective`, directly rather than through the parameter grid. The first two allow us to access GPU capabilities directly, `verbosity` gives us visibility as to what the model is

running in real-time, and `eval_metric` is the evaluation metric to be used for validation data—as you can see, you can pass multiple evaluation metrics in the form of a `Python list`.

Unfortunately, using `tree_method` and `predictor` we kept getting the same error over and over. You can track the status of this error here.

```
Kernel error:
In: /workspace/include/xgboost/../../../src/common/span.h,  line: 489
 T &xgboost::common::Span<T, Extent>::operator[](long) const [with T =
xgboost::detail::GradientPairInternal<float>, Extent = -1L]
 Expecting: _idx >= 0 && _idx < size()
terminate called after throwing an instance of 'thrust::system::system_error'
  what():  function_attributes(): after cudaFuncGetAttributes: unspecified launch failure
```

Given that we couldn't fix the root issue (**pun intended**), the model had to be run on CPU. So the final code for our `RandomizedSearchCV` looked like this:

```
# Create XGB Classifier object
xgb_clf = xgb.XGBClassifier(tree_method = "exact", predictor = "cpu_predictor", verbosity =
                            objective = "multi:softmax")

# Create parameter grid
parameters = {"learning_rate": [0.1, 0.01, 0.001],
              "gamma" : [0.01, 0.1, 0.3, 0.5, 1, 1.5, 2],
              "max_depth": [2, 4, 7, 10],
              "colsample_bytree": [0.3, 0.6, 0.8, 1.0],
              "subsample": [0.2, 0.4, 0.5, 0.6, 0.7],
              "reg_alpha": [0, 0.5, 1],
```

`RandomizedSearchCV` allows us to find the best combination of hyperparameters from the options given of the parameter grid. We can then access these through `model_xgboost.best_estimator_.get_params()` so we can use them on the next iteration of the model. Below are the best estimators for this model.

```
Learning Rate: 0.1
Gamma: 0.1
Max Depth: 4
Subsample: 0.7
Max Features at Split: 1
Alpha: 0
Lambda: 1
Minimum Sum of the Instance Weight Hessian to Make Child: 7
Number of Trees: 100
Accuracy Score: 0.883
```

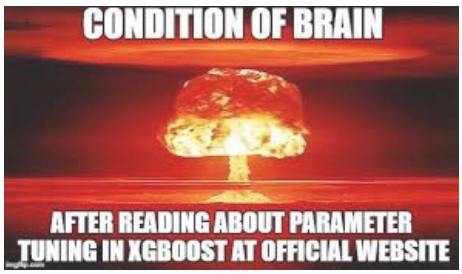
Our accuracy score went up by 2%! Not bad for a model that was running for over 60 hours!

Now, let's take a look at each hyperparameter individually.

- `learning_rate`: to start, let's clarify that this learning rate is not the same as in gradient descent. In the case of gradient boosting, the learning rate is meant to

lessen the effect of each additional tree to the model. In their paper, *A Scalable Tree Boosting System* Tianqi Chen and Carlos Guestrin refer to this regularization technique as *shrinkage*, and it is an additional method to prevent overfitting. The lower the learning rate, the more robust the model will be in preventing overfitting.

- `gamma` : mathematically, this is known as the *Lagrangian Multiplier*, and its purpose is complexity control. It is a pseudo-regularization term for the loss function; and it represents by how much the loss has to be reduced when considering a split, in order for that split to happen.
- `max_depth` : refers to the depth of a tree. It sets the maximum number of nodes that can exist between the root and the farthest leaf. Remember that deeper trees are prone to overfitting.
- `colsample_bytreee` : represents a fraction of the columns (*features*) to be considered at each split. It is referred to in the paper *A Scalable Tree Boosting System by* Tianqi Chen and Carlos Guestrin as another of the main techniques to prevent overfitting and to improve the computational speed.
- `subsample` : represents a fraction of the rows (*observations*) to be considered when building each subtree. Tianqi Chen and Carlos Guestrin in their paper *A Scalable Tree Boosting System* recommend `colsample_bytree` over `subsample` to prevent overfitting, as they found that the former is more effective for this purpose.
- `reg_alpha` : L1 regularization term. L1 regularization encourages sparsity (meaning pulling weights to 0). It can be more useful when the `objective` is logistic regression since you might need help with feature selection.
- `reg_lambda` : L2 regularization term. L2 encourages smaller weights, this approach can be more useful in tree-models where zeroing features might not make much sense.
- `min_child_weight` : similar to `gamma` , as it performs regularization at the splitting step. It is the minimum Hessian weight required to create a new node. The Hessian is the second derivative.
- `n_estimators` : the number of trees to fit.



The Ugly Truth

But wait! There's more. There are other hyperparameters you can modify, depending on the problem you are trying to solve or what you're trying to optimize for:

- `booster` : allows you to choose which booster to use: `gbtree` , `gblinear` or `dart` . We've been using `gbtree` , but `dart` and `gblinear` also have their own additional hyperparameters to explore.

- `scale_pos_weight` : balances between negative and positive weights, and should definitely be used in cases where the data present high class imbalance.
- `importance_type` : refers to the feature importance type to be used by the `feature_importances_` method. `gain` calculates the relative contribution of a feature to all the trees in a model (the higher the relative gain, the more relevant the feature). `cover` calculates the relative number of observations related to a feature when used to decide the leaf node. `weight` measures the relative number of times a feature is used to split the data across all the trees in a model.
- `base_score` : global bias. This parameter is useful when dealing with high class imbalance.
- `max_delta_step` : sets the maximum absolute value possible for the weights. Also useful when dealing with unbalanced classes.

Note: we performed `RandomizedSearchCV` using the Scikit-Learn wrapper interface for XGBoost.

What comes next

Once you have your model's best estimators, you could do a number of different things. My recommendation would be to use them as hyperparameters on XGBoost's built-in Cross Validation so you can make use of the `early_stopping_rounds` functionality—another step towards optimization and overfit prevention. But that's a story for another day!

Machine Learning | Data Science | Classification | Xgboost | Python

Florencia Leoni
Medium member since Mar 2019
Lawyer and Economist turned Data Scientist. Passionately curious, avid notetaker, and impulsive book buyer. Living for the love of knowledge.

Follow

Towards Data Science
Sharing concepts, ideas, and codes.

Follow