Optimizing XGBoost: A Guide to Hyperparameter Tuning

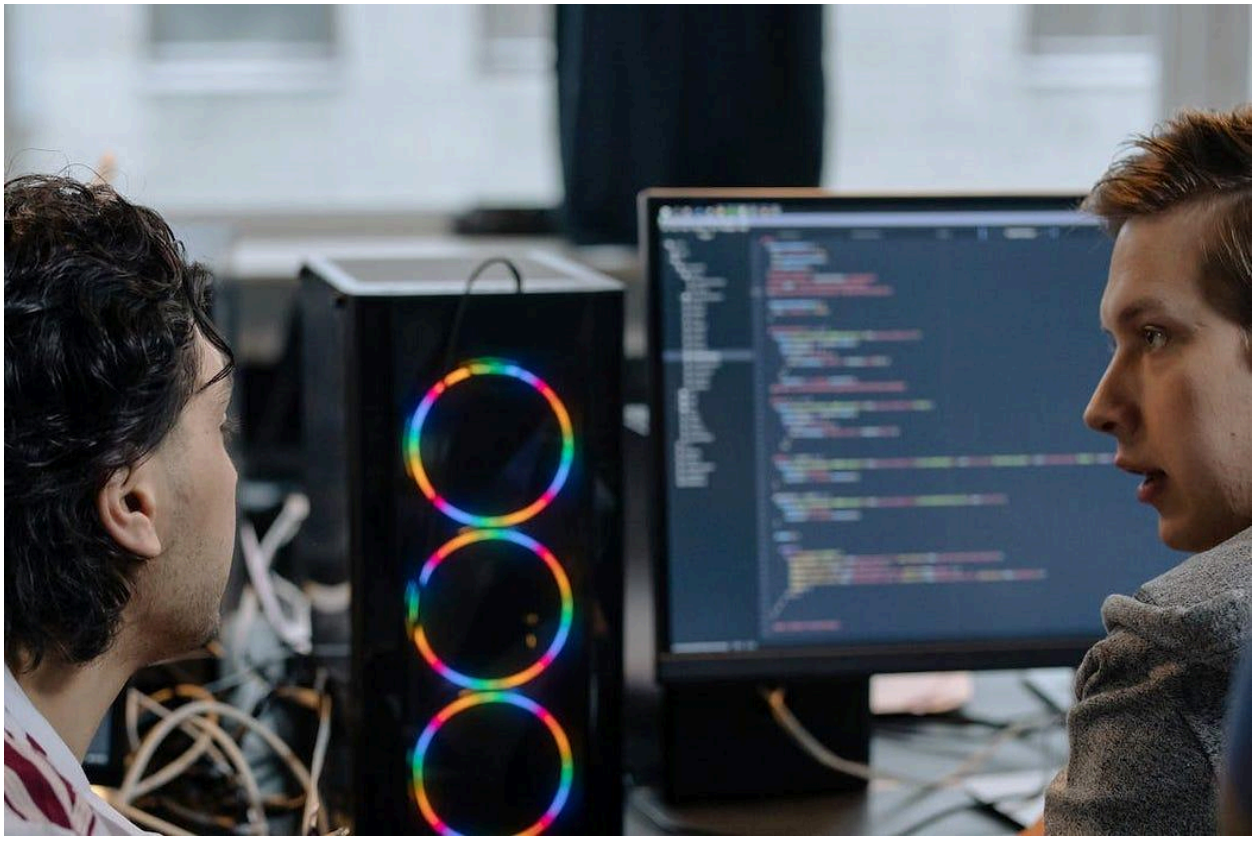
RITHP

Follow

4 min read

.

Jan 17, 2023



Hyperparameter tuning is important because the performance of a machine learning model is heavily influenced by the choice of hyperparameters. Choosing the right set of hyperparameters can lead to better model performance, while choosing the wrong set can lead to poor performance. Additionally, when a model has too many hyperparameters, it can be difficult or even impossible to find the best set of hyperparameters manually.

Different types of hyperparameters in XGBoost

In XGBoost, there are two main types of hyperparameters: tree-specific and learning task-specific.

Tree-specific hyperparameters control the construction and complexity of the decision trees:

- `max_depth`: maximum depth of a tree. Deeper trees can capture more complex patterns in the data, but may also lead to overfitting.
- `min_child_weight`: minimum sum of instance weight (hessian) needed in a child. This can be used to control the complexity of the decision tree by preventing the creation of too small leaves.
- `subsample`: percentage of rows used for each tree construction. Lowering this value can prevent overfitting by training on a smaller subset of the data.

- `colsample_bytree`: percentage of columns used for each tree construction. Lowering this value can prevent overfitting by training on a subset of the features.

Learning task-specific hyperparameters control the overall behavior of the model and the learning process:

- `eta` (also known as learning rate): step size shrinkage used in updates to prevent overfitting. Lower values make the model more robust by taking smaller steps.
- `gamma`: minimum loss reduction required to make a further partition on a leaf node of the tree. Higher values increase the regularization.
- `lambda`: L2 regularization term on weights. Higher values increase the regularization.

- `alpha`: L1 regularization term on weights. Higher values increase the regularization.

Overview of different techniques for tuning hyperparameters

Grid search is one of the most widely used techniques for hyperparameter tuning. It involves specifying a set of possible values for each hyperparameter, and then training and evaluating the model for each combination of hyperparameter values. Grid search is simple to implement and can be efficient when the number of hyperparameters and their possible values is small. However, it can become computationally expensive as the number of hyperparameters and possible values increases.

```
import xgboost as xgb
```

```
from sklearn.model_selection import GridSearchCV
```

```
# Define the hyperparameter grid

param_grid = {

    'max_depth': [3, 5, 7],

    'learning_rate': [0.1, 0.01, 0.001],

    'subsample': [0.5, 0.7, 1]

}




# Create the XGBoost model object

xgb_model = xgb.XGBClassifier()




# Create the GridSearchCV object
```

```
grid_search = GridSearchCV(xgb_model, param_grid, cv=5,
scoring='accuracy')
```

```
# Fit the GridSearchCV object to the training data
```

```
grid_search.fit(X_train, y_train)
```

```
# Print the best set of hyperparameters and the corresponding score
```

```
print("Best set of hyperparameters: ", grid_search.best_params_)
```

```
print("Best score: ", grid_search.best_score_)
```

Random search is a variation of grid search that randomly samples from the set of possible hyperparameter values instead of trying all combinations. This can be more efficient than grid search because it does not need to evaluate all possible combinations. However, it can

still be computationally expensive, especially when the number of hyperparameters and possible values is large.

```
import xgboost as xgb

from sklearn.model_selection import RandomizedSearchCV

import scipy.stats as stats



# Define the hyperparameter distributions

param_dist = {

    'max_depth': stats.randint(3, 10),

    'learning_rate': stats.uniform(0.01, 0.1),

    'subsample': stats.uniform(0.5, 0.5),

    'n_estimators':stats.randint(50, 200)
```

```
}
```

```
# Create the XGBoost model object
```

```
xgb_model = xgb.XGBClassifier()
```

```
# Create the RandomizedSearchCV object
```

```
random_search = RandomizedSearchCV(xgb_model,
param_distributions=param_dist, n_iter=10, cv=5, scoring='accuracy')
```

```
# Fit the RandomizedSearchCV object to the training data
```

```
random_search.fit(X_train, y_train)
```

```
# Print the best set of hyperparameters and the corresponding score

print("Best set of hyperparameters: ", random_search.best_params_)

print("Best score: ", random_search.best_score_)
```

Bayesian optimization is a more sophisticated technique that uses Bayesian methods to model the underlying function that maps hyperparameters to the model performance. It tries to find the optimal set of hyperparameters by making smart guesses based on the previous results. Bayesian optimization is more efficient than grid or random search because it attempts to balance exploration and exploitation of the search space. It can also deal with the cases of large number of hyperparameters and large search space. However, it can be more difficult to implement than grid search or random search and may require more computational resources.

```
import xgboost as xgb

from hyperopt import fmin, tpe, hp
```

```
# Define the hyperparameter space

space = {

    'max_depth': hp.quniform('max_depth', 2, 8, 1),

    'learning_rate': hp.loguniform('learning_rate', -5, -2),

    'subsample': hp.uniform('subsample', 0.5, 1)

}



# Define the objective function to minimize

def objective(params):

    xgb_model = xgb.XGBClassifier(**params)

    xgb_model.fit(X_train, y_train)
```

```
    y_pred = xgb_model.predict(X_test)

    score = accuracy_score(y_test, y_pred)

    return {'loss': -score, 'status': STATUS_OK}



# Perform the optimization

best_params = fmin(objective, space, algo=tpe.suggest, max_evals=100)

print("Best set of hyperparameters: ", best_params)
```

There are several techniques that can be used to tune the hyperparameters of an XGBoost model including grid search, random search and Bayesian optimization. Grid search is simple to implement but can be computationally expensive when the number of hyperparameters and possible values is large. Random search is more efficient than grid search but still can be computationally expensive. Bayesian optimization is the most sophisticated technique, which

balances exploration and exploitation, but can be more difficult to implement and require more computational resources.