xgboost-3-model-tuning

May 9, 2019

1 Modeling tuning in XGBoost

Miao Cai

When performing parameter searches, we will use a dictionary that we typically call a parameter grid, because it will contain ranges of values over which we will search to find an optimal configuration.

Important parameters in XGBoost:

- "objective": "reg:linear"
- "colsample_bytree": 0.3
- "learning rate": 0.1
- "max_depth": 5

Instead of attempting to cherry pick the best possible number of boosting rounds, you can very easily have XGBoost automatically select the number of boosting rounds for you within `xgb.cv()`. This is done using a technique called **early stopping**.

Early stopping works by testing the XGBoost model after every boosting round against a hold-out dataset and stopping the creation of additional boosting rounds (thereby finishing training of the model early) if the hold-out metric ("rmse" in our case) does not improve for a given number of rounds.

Here you will use the `early_stopping_rounds` parameter in `xgb.cv()` with a large possible number of boosting rounds (50). Bear in mind that if the holdout metric continuously improves up through when `num_boosting_rounds` is reached, then early stopping does not occur.

1.1 XGBoost hyperparameters

Common tree tunable parameters

- learning rate (`eta`): how quickly the model fits the residual error using additional base learners. A low learning rate will require more boosting rounds to achieve the same reduction in residual error as an XGBoost model with a high learning rate. A high learning rate will penalize feature weights more strongly, causing much stronger regularization.
- gamma: min loss reduction to create new tree split
- lambda: L2 reg on leaf weights
- alpha: L1 reg on leaf weights
- max_depth: max depth per tree

- subsample: % sample used per tree (between 0 and 1). If the value is low, the fraction of your training data used would per boosting round would be low, and you may run into underfitting problems. A value that is very high can lead to overfitting as well.
- colsample_bytree: the fraction of features you can select from during any given boosting round and must also be a value between 0 and 1. A large value means that almost all features can be used to build a tree during a given boosting, whereas a small value means that the fraction of features that can be selected from is very small. In general, smaller colsample_bytree values can be thought of as providing additional regularization to the model, whereas using all columns may in certain cases overfit a trained model.

gamma, lambda, and alpha will have an effect on how strongly regularized the trained model will be.

Linear tunable parameters

The number of tunable parameters is significantly smaller.

- lambda: L2 reg on weights
- alpha: L1 reg on weights
- lambda_bias: L2 reg term on bias

You can also tune the number of estimators used for both base model types, which is hte number of boosting rounds (either the number of trees you build or the number of linear base learners you construct)

```
In [1]: # Create your housing DMatrix: housing_dmatrix
        #  housing_dmatrix = xgb.DMatrix(data=X, label=y)

        # Create the parameter dictionary for each tree (boosting round)
        #  params = {"objective":"reg:linear", "max_depth":3}

        # Create list of eta values and empty list to store final round rmse per xgboost model
        #  eta_vals = [0.001, 0.01, 0.1]
        #  best_rmse = []

        # Systematically vary the eta
        #  for curr_val in eta_vals:

        #      params["eta"] = curr_val

            # Perform cross-validation: cv_results
        #      cv_results = xgb.cv(dtrain=housing_dmatrix, params=params,
        #          early_stopping_rounds=5, num_boost_round=10, metrics="rmse", seed=123)



            # Append the final round rmse to best_rmse
        #      best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

        # Print the resultant DataFrame
        #  print(pd.DataFrame(list(zip(eta_vals, best_rmse)), columns=["eta","best_rmse"]))
```

1.2 Grid search and random search

Grid search: a method of exhausively searching through a collection of possible parameter values.

- The number of models = number of distinct values per hyperparameter multiplied across each hyperparameter
- Pick final model hyperparameter values that give best cross-validated evaluation metric value

Random search: simply involves drawing a random combination of possible hyperparameter values from the range of allowable hyperparameters a set number of times.

- Create a (possible infinite) range of hyperparameter values per hyperparameter that you would like to search over.
- Set the number of iterations you would like for the random search to continue
- During each iteration, randomly draw a value in the range of specified values for each hyperparameter searched over and train/evaluate a model with those hyperparameters.
- After you've reached the maximum number of iterations, select the hyperparameter configuration with the best evaluated score.

```
In [2]: # Create your housing DMatrix: housing_dmatrix
        #  housing_dmatrix = xgb.DMatrix(data=X, label=y)

        # Create the parameter grid: gbm_param_grid
        #  gbm_param_grid = {
        #      'colsample_bytree': [0.3, 0.7],
        #      'n_estimators': [50],
        #      'max_depth': [2, 5]
        #  }

        # Instantiate the regressor: gbm
        #  gbm = xgb.XGBRegressor()

        # Perform grid search: grid_mse
        #  grid_mse = GridSearchCV(estimator=gbm,
        #  param_grid=gbm_param_grid,
        #  scoring='neg_mean_squared_error', cv=4, verbose=1)


        # Fit grid_mse to the data
        #  grid_mse.fit(X, y)

        # Print the best parameters and lowest RMSE
        #  print("Best parameters found: ", grid_mse.best_params_)
        #  print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
```

1.3 The limites of grid search and random search

Grid search:

- Number of models you must build with every additional new parameter grows very quickly

Random search

- Parameter space to explore can be massive
- Randomly jumping throughout the space looking for a "best" result becomes a waiting game.

You can always increase the number of iterations you want the random search to run, but then finding an optimal configuration becomes a combination of waiting randomly finding a good set of hyperparameters.

The search space size can be massive for Grid Search in certain cases, whereas for Random Search the number of hyperparameters has a significant effect on how long it takes to run.