A Guide on XGBoost hyperparameters tuning_

Hello friends,

In my previous kernel XGBoost + k-fold CV + Feature Importance, we have discussed XGBoost and develop a simple baseline XGBoost model.

Now, XGBoost algorithm provides large range of hyperparameters. We should know how to tune these hyperparameters to improve and take full advantage of the XGBoost model.

Hence, in this kernel, we will discuss main hyperparameters of the XGBoost model and how to tune these hyperparameters.

So, let's get started.

If this helped in your learning, then please UPVOTE – as they are the source of motivation!

Happy Learning

Table of Contents

1. What are hyperparameters

Table of Contents

- In this kernel, we will discuss the critical problem of hyperparameter tuning in XGBoost model.
- **Hyperparameters** are certain values or weights that determine the learning process of an algorithm.
- As stated earlier, XGBoost provides large range of hyperparameters. We can leverage the maximum power of XGBoost by tuning its hyperparameters.
- The most powerful ML algorithm like XGBoost is famous for picking up patterns and regularities in the data by automatically tuning thousands of learnable parameters.
- In tree-based models, like XGBoost the learnable parameters are the choice of decision variables at each node.
- XGBoost is a very powerful algorithm. So, it will have more design decisions and hence large hyperparameters. These are parameters specified by hand to the algo and fixed throughout a training phase.
- In tree-based models, hyperparameters include things like the maximum depth of the tree, the number of trees to grow, the number of variables to consider when building each tree, the minimum number of samples on a leaf and the fraction of observations used to build a tree.

- Although we focus on optimizing XGBoost hyperparameters in this kernel, the concepts discussed in this kernel applies to any other advanced ML algorithm as well.

2. XGBoost hyperparameters

Table of Contents

- Generally, the XGBoost hyperparameters have been divided into 4 categories. They are as follows -
    1. General parameters
    2. Booster parameters
    3. Learning task parameters
    4. Command line parameters
- Before running a XGBoost model, we must set three types of parameters - **general parameters**, **booster parameters** and **task parameters**.
- The fourth type of parameters are **command line parameters**. They are only used in the console version of XGBoost. So, we will skip these parameters and limit our discussion to the first three type of parameters.

2.1 General Parameters

Table of Contents

- These parameters guide the overall functioning of the XGBoost model.
- In this section, we will discuss three hyperparameters - **booster**, **verbosity** and **nthread**.
- Please visit XGBoost General Parameters for detailed discussion on general parameters.

2.1.1 booster

Table of Contents

- **booster[default = gbtree]**
    - **booster** parameter helps us to choose which booster to use.
    - It helps us to select the type of model to run at each iteration.
    - It has 3 options - **gbtree**, **gblinear** or **dart**.
        - **gbtree** and **dart** - use tree-based models, while
        - **gblinear** uses linear models.

2.1.2 verbosity

Table of Contents

- **verbosity[default = 1]**
  - Verbosity of printing messages.
  - Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug).

2.1.3 nthread

Table of Contents

- **nthread [default = maximum number of threads available if not set]**
  - This is number of parallel threads used to run XGBoost.
  - This is used for parallel processing and number of cores in the system should be entered.
  - If you wish to run on all cores, value should not be entered and algorithm will detect automatically.

There are other general parameters like **disable_default_eval_metric [default=0]**, **num_pbuffer [set automatically by XGBoost, no need to be set by user]** and **num_feature [set automatically by XGBoost, no need to be set by user]**.

So, these parameters are taken care by XGBoost algorithm itself. Hence,we will not discuss these further.

2.2 Booster Parameters

Table of Contents

- We have 2 types of boosters - **tree booster** and **linear booster**.
- We will limit our discussion to **tree booster** because it always outperforms the **linear booster** and thus the later is rarely used.
- Please visit, Parameters for Tree Booster, for detailed discussion on booster parameters.

2.2.1 eta

Table of Contents

- **eta [default=0.3, alias: learning_rate]**
  - It is analogous to learning rate in GBM.
  - It is the step size shrinkage used in update to prevent overfitting.
  - After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
  - It makes the model more robust by shrinking the weights on each step.
  - range : [0,1]
  - Typical final values : 0.01-0.2.

2.2.2 gamma

Table of Contents

- **gamma [default=0, alias: min_split_loss]**
  - A node is split only when the resulting split gives a positive reduction in the loss function.
  - Gamma specifies the minimum loss reduction required to make a split.
  - It makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
  - The larger gamma is, the more conservative the algorithm will be.
  - Range: [0,∞]

2.2.3 max_depth

Table of Contents

- **max_depth [default=6]**
  - The maximum depth of a tree, same as GBM.
  - It is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
  - Increasing this value will make the model more complex and more likely to overfit.
  - The value 0 is only accepted in lossguided growing policy when tree_method is set as hist and it indicates no limit on depth.
  - We should be careful when setting large value of max_depth because XGBoost aggressively consumes memory when training a deep tree.
  - range: [0,∞] (0 is only accepted in lossguided growing policy when tree_method is set as hist.
  - Should be tuned using CV.

    - Typical values: 3-10

2.2.4 min_child_weight

Table of Contents

- **min_child_weight [default=1]**
    - It defines the minimum sum of weights of all observations required in a child.
    - This is similar to min_child_leaf in GBM but not exactly. This refers to min “sum of weights” of observations while GBM has min “number of observations”.
    - It is used to control over-fitting.
    - Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
    - Too high values can lead to under-fitting.
    - Hence, it should be tuned using CV.
    - The larger min_child_weight is, the more conservative the algorithm will be.
    - range: [0,∞]

2.2.5 max_delta_step

Table of Contents

- **max_delta_step [default=0]**
    - In maximum delta step we allow each tree’s weight estimation to be.
    - If the value is set to 0, it means there is no constraint.
    - If it is set to a positive value, it can help making the update step more conservative.
    - Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced.
    - Set it to value of 1-10 might help control the update.
    - range: [0,∞]

2.2.6 subsample

Table of Contents

- **subsample [default=1]**
    - It denotes the fraction of observations to be randomly samples for each tree.
    - Subsample ratio of the training instances.

    - Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. - This will prevent overfitting.
    - Subsampling will occur once in every boosting iteration.
    - Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
    - Typical values: 0.5-1
    - range: (0,1]

2.2.7 colsample_bytree, colsample_bylevel, colsample_bynode

Table of Contents

- **colsample_bytree, colsample_bylevel, colsample_bynode [default=1]**
    - This is a family of parameters for subsampling of columns.
    - All **colsample_by** parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled.
    - **colsample_bytree** is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.
    - **colsample_bylevel** is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.
    - **colsample_bynode** is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.
    - **colsample_by*** parameters work cumulatively. For instance, the combination **{'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5}** with 64 features will leave 8 features to choose from at each split.

2.2.8 lambda

Table of Contents

- **lambda [default=1, alias: reg_lambda]**
    - L2 regularization term on weights (analogous to Ridge regression).
    - This is used to handle the regularization part of XGBoost.
    - Increasing this value will make model more conservative.

2.2.9 alpha

Table of Contents

- **alpha [default=0, alias: reg_alpha]**
    - L1 regularization term on weights (analogous to Lasso regression).
    - It can be used in case of very high dimensionality so that the algorithm runs faster when implemented.
    - Increasing this value will make model more conservative.

2.2.10 tree_method

Table of Contents

- **tree_method string [default= auto]**
    - The tree construction algorithm used in XGBoost.
    - XGBoost supports `approx`, `hist` and `gpu_hist` for distributed training. Experimental support for external memory is available for `approx` and `gpu_hist`.
    - Choices: `auto`, `exact`, `approx`, `hist`, `gpu_hist`
        - **auto**: Use heuristic to choose the fastest method.
            - For small to medium dataset, exact greedy (exact) will be used.
            - For very large dataset, approximate algorithm (approx) will be chosen.
            - Because old behavior is always use exact greedy in single machine, user will get a message when approximate algorithm is chosen to notify this choice.
        - **exact**: Exact greedy algorithm.
        - **approx**: Approximate greedy algorithm using quantile sketch and gradient histogram.
        - **hist**: Fast histogram optimized approximate greedy algorithm. It uses some performance improvements such as bins caching.
        - **gpu_hist**: GPU implementation of hist algorithm.

2.2.11 scale_pos_weight

Table of Contents

- **scale_pos_weight [default=1]**
    - It controls the balance of positive and negative weights,
    - It is useful for imbalanced classes.
    - A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

    - A typical value to consider: `sum(negative instances) / sum(positive instances)`.

2.2.12 max_leaves

Table of Contents

- **max_leaves [default=0]**
    - Maximum number of nodes to be added.
    - Only relevant when `grow_policy=lossguide` is set.
- There are other hyperparameters like `sketch_eps`,`updater`, `refresh_leaf`, `process_type`, `grow_policy`, `max_bin`, `predictor` and `num_parallel_tree`.
- For detailed discussion of these hyperparameters, please visit Parameters for Tree Booster

2.3 Learning Task Parameters

Table of Contents

- These parameters are used to define the optimization objective the metric to be calculated at each step.
- They are used to specify the learning task and the corresponding learning objective. The objective options are below:

2.3.1 objective

Table of Contents

- **objective [default=reg:squarederror]**
- It defines the loss function to be minimized. Most commonly used values are given below -
    - **reg:squarederror** : regression with squared loss.
    - **reg:squaredlogerror**: regression with squared log loss 1/2[log(pred+1)−log(label+1)]2. - All input labels are required to be greater than -1.
    - **reg:logistic** : logistic regression
    - **binary:logistic** : logistic regression for binary classification, output probability
    - **binary:logitraw**: logistic regression for binary classification, output score before logistic transformation
    - **binary:hinge** : hinge loss for binary classification. This makes predictions of 0 or 1, rather than producing probabilities.
    - **multi:softmax** : set XGBoost to do multiclass classification using the softmax objective, you also need to set num_class(number of classes)

- **multi:softprob** : same as softmax, but output a vector of ndata *nclass, which can be further reshaped to ndata* nclass matrix. The result contains predicted probability of each data point belonging to each class.

2.3.2 eval_metric

Table of Contents

- **eval_metric [default according to objective]**
- The metric to be used for validation data.
- The default values are **rmse for regression**, **error for classification** and **mean average precision for ranking**.
- We can add multiple evaluation metrics.
- Python users must pass the metrices as list of parameters pairs instead of map.
- The most common values are given below -
  - **rmse** : root mean square error
  - **mae** : mean absolute error
  - **logloss** : negative log-likelihood
  - **error** : Binary classification error rate (0.5 threshold). It is calculated as `#(wrong cases)/#(all cases)`. For the predictions, the evaluation will regard the instances with prediction value larger than 0.5 as positive instances, and the others as negative instances.
  - **merror** : Multiclass classification error rate. It is calculated as `#(wrong cases)/#(all cases)`.
  - **mlogloss** : Multiclass logloss
  - **auc**: Area under the curve
  - **aucpr** : Area under the PR curve

2.3.3 seed

Table of Contents

- **seed [default=0]**
  - The random number seed.
  - This parameter is ignored in R package, use set.seed() instead.
  - It can be used for generating reproducible results and also for parameter tuning.

3. Basic Setup

Table of Contents

3.1 Import libraries

Table of Contents

```
# import pandas for data wrangling
import pandas as pd


# import numpy for Scientific computations
import numpy as np


# import machine learning libraries
import xgboost as xgb
from sklearn.metrics import accuracy_score


# import packages for hyperparameters tuning
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe


# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will
list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Any results you write to the current directory are saved as output.
```

```
/kaggle/input/wholesale-customers-data-set/Wholesale customers data.csv
```

3.2 Read dataset

Table of Contents

```
data = '/kaggle/input/wholesale-customers-data-set/Wholesale customers data.csv'

df = pd.read_csv(data)
```

I will skip the EDA part, as I have done it in previous kernel - XGBoost + k-fold CV + Feature Importance.

3.3 Declare feature vector and target variable

Table of Contents

```
X = df.drop('Channel', axis=1)

y = df['Channel']
```

- Now, let's take a look at feature vector(X) and target variable(y).

```
X.head()
```

	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	3	12669	9656	7561	214	2674	1338

1	3	7057	9810	9568	1762	3293	1776
2	3	6353	8808	7684	2405	3516	7844
3	3	13265	1196	4221	6404	507	1788
4	3	22615	5410	7198	3915	1777	5185

```
y.head()
```

```
0    2
1    2
2    2
3    1
4    2

Name: Channel, dtype: int64
```

- We can see that the y label contain values as 1 and 2.
- We will need to convert it into 0 and 1 for further analysis.
- We will do it as follows -

```
# convert labels into binary values

y[y == 2] = 0

y[y == 1] = 1
```

```
# again preview the y label
```

```
y.head()
```

```
0    0
1    0
2    0
3    1
4    0

Name: Channel, dtype: int64
```

We can see that our target variable (y) has been converted into 0 and 1.

3.4 Split data into separate training and test set

Table of Contents

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
random_state = 0)
```

4. Bayesian Optimization with HYPEROPT

Table of Contents

- **Bayesian optimization** is optimization or finding the best parameter for a machine learning or deep learning algorithm.
- **Optimization** is the process of finding a minimum of cost function , that determines an overall better performance of a model on both train-set and test-set.
- In this process, we train the model with various possible range of parameters until a best fit model is obtained.
- **Hyperparameter tuning** helps in determining the optimal tuned parameters and return the best fit model, which is the best practice to follow while building an ML or DL model.
- In this section, we discuss one of the most accurate and successful hyperparameter tuning method, which is **Bayesian Optimization with HYPEROPT**.

- Please see my kernel Bayesian Optimization using HYPEROPT, for more information on the optimization process using HYPEROPT.
- So, we will start with **HYPEROPT**.

4.1 What is HYPEROPT

Table of Contents

- **HYPEROPT** is a powerful python library that search through an hyperparameter space of values and find the best possible values that yield the minimum of the loss function.
- Bayesian Optimization technique uses Hyperopt to tune the model hyperparameters. Hyperopt is a Python library which is used to tune model hyperparameters.
- More information on Hyperopt can be found at the following link:-

https://hyperopt.github.io/hyperopt/?source=post_page

4.2 4 parts of Optimization Process

Table of Contents

The optimization process consists of 4 parts which are as follows-

- **1. Initialize domain space**

The domain space is the input values over which we want to search.

- **2. Define objective function**

The objective function can be any function which returns a real value that we want to minimize. In this case, we want to minimize the validation error of a machine learning model with respect to the hyperparameters. If the real value is accuracy, then we want to maximize it. Then the function should return the negative of that metric.

- **3. Optimization algorithm**

It is the method used to construct the surrogate objective function and choose the next values to evaluate.

- **4. Results**

Results are score or value pairs that the algorithm uses to build the model.

4.3 Bayesian Optimization implementation

Table of Contents

4.3.1 Initialize domain space for range of values

Table of Contents

```
space={'max_depth': hp.quniform("max_depth", 3, 18, 1),
        'gamma': hp.uniform ('gamma', 1,9),
        'reg_alpha' : hp.quniform('reg_alpha', 40,180,1),
        'reg_lambda' : hp.uniform('reg_lambda', 0,1),
        'colsample_bytree' : hp.uniform('colsample_bytree', 0.5,1),
        'min_child_weight' : hp.quniform('min_child_weight', 0, 10, 1),
        'n_estimators': 180,
        'seed': 0
    }
```

The available hyperopt optimization algorithms are -

- **hp.choice(label, options)** — Returns one of the options, which should be a list or tuple.
- **hp.randint(label, upper)** — Returns a random integer between the range [0, upper).
- **hp.uniform(label, low, high)** — Returns a value uniformly between low and high.
- **hp.quniform(label, low, high, q)** — Returns a value round(uniform(low, high) / q) * q, i.e it rounds the decimal values and returns an integer.
- **hp.normal(label, mean, std)** — Returns a real value that's normally-distributed with mean and standard deviation sigma.

4.3.2 Define objective function

Table of Contents

```
def objective(space):
    clf=xgb.XGBClassifier(
```

```
                    n_estimators =space['n_estimators'], max_depth =
int(space['max_depth']), gamma = space['gamma'],
                    reg_alpha =
int(space['reg_alpha']),min_child_weight=int(space['min_child_weight']),
                    colsample_bytree=int(space['colsample_bytree']))

    evaluation = [( X_train, y_train), ( X_test, y_test)]

    clf.fit(X_train, y_train,
            eval_set=evaluation, eval_metric="auc",
            early_stopping_rounds=10,verbose=False)


    pred = clf.predict(X_test)
    accuracy = accuracy_score(y_test, pred>0.5)
    print ("SCORE:", accuracy)
    return {'loss': -accuracy, 'status': STATUS_OK }
```

4.3.3 Optimization algorithm

Table of Contents

```
trials = Trials()

best_hyperparams = fmin(fn = objective,
                        space = space,
                        algo = tpe.suggest,
                        max_evals = 100,
                        trials = trials)
```

```
SCORE:
0.3484848484848485

SCORE:
0.8712121212121212
```

```
100%|██████████| 100/100 [00:06<00:00, 15.09it/s, best loss:
-0.8712121212121212]
```

- Here **best_hyperparams** gives us the optimal parameters that best fit model and better loss function value.
- **trials** is an object that contains or stores all the relevant information such as hyperparameter, loss-functions for each set of parameters that the model has been trained.
- **'fmin'** is an optimization function that minimizes the loss function and takes in 4 inputs - fn, space, algo and max_evals.
- Algorithm used is **tpe.suggest**.

4.3.4 Print Results

Table of Contents

```
print("The best hyperparameters are : ","\n")
print(best_hyperparams)
```

```
The best hyperparameters are :

{'colsample_bytree': 0.6655392754230048, 'gamma': 4.198875359789924,
'max_depth': 17.0, 'min_child_weight': 1.0, 'reg_alpha': 57.0,
'reg_lambda': 0.896332305739873}
```

- The above result give best set of hyperparameters.

5. Results and Conclusion

Table of Contents

- In this kernel, we have discussed the XGBoost hyperparameters which are divided into 3 categories - general parameters, booster parameters and learning task parameters.
- We have discussed **Bayesian Optimization with HYPEROPT**.
- We have discussed the 4 parts of optimization process.
- We have found the best hyperparameters for the XGBoost ML model.

- The same technique can be applied to find the optimum hyperparameters for any other ML model.

6. References

Table of Contents

The ideas and concepts in this kernel are taken from the following websites.

- https://xgboost.readthedocs.io/en/latest/tutorials/param_tuning.html
- https://xgboost.readthedocs.io/en/latest/parameter.html#general-parameters
- https://medium.com/analytics-vidhya/hyperparameter-tuning-hyperopt-bayesian-optimization-for-xgboost-and-neural-network-8aedf278a1c9
- https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
- https://www.kaggle.com/yassinealouini/hyperopt-the-xgboost-model

So, now we will come to the end of this kernel.

I hope you find this kernel useful and enjoyable.

Your comments and feedback are most welcome.

Thank you