Hyperparameter tuning in XGBoost

Cambridge Spark Follow
Oct 9, 2017 · 15 min read

This tutorial is the second part of our series on XGBoost. If you haven't done it yet, for an introduction to XGBoost check Getting started with XGBoost.

With this tutorial you will learn to use the native XGBoost API (for the sklearn API, see the previous tutorial) that comes with its own cross-validation and other nice features. You will learn the role of the main hyperpameters and techniques to tune your model.

Other topics that you will come across in this tutorial include:

- Tuning XGboost hyperparameters
- Using a `watchlist` and `early_stopping_round` with XGBoost's native API
- DMatrices (XGBoost data format)
- Bias and variance trade off
- Timing in a Jupyter notebook
- Cross-validation
- Using a baseline model
- Mean Absoluate Error
- Grid Search
- Saving and loading an XGboost model

Let's start with a short introduction to the XGBoost native API.

The native XGBoost API

Although the scikit-learn API of XGBoost (shown in the previous tutorial) is easy to use and fits well in a scikit-learn pipeline, it is sometimes better to use the native API. Advantages include:

- Automatically find the best number of boosting rounds
- Built-in cross validation
- Custom objective functions

Find more details online.

DMatrices

Instead of numpy arrays or pandas dataFrame, XGBoost uses DMatrices. A DMatrix can contain both the features and the target. If you already have loaded you data into numpy arrays `x` and `y`, you can create a DMatrix with:

```
xgb.DMatrix(X, label=y)
```

To read more about DMatrices check the documentation.

Data/problem

We will solve a regression problem here, but what you will learn is also applicable to classification. Download the dataset and unzip it.

This dataset is composed of 53 features describing a post on Facebook: the number of likes on the page it was posted, the category of the page, the time and day it was posted, etc. The last column is the target: the number of comments the post received. Our goal is to predict the number of

comments a new post will receive based on all the given features.

First make sure you install the libraries we will use for this tutorial. You need to install XGBoos, pandas and numpy. If you are using `pip`, you can do it by executing the following command in your notebook:

```
!pip install xgboost scikit-learn pandas numpy
```

If you experience issues installing XGBoost with `pip`, check our previous tutorial

Load the dataset with pandas

```
import pandas as pd
file = "datasets/facebook_comments/Dataset/Training
/Features_Variant_1.csv"
df = pd.read_csv(file, header=None)
df.sample(n=5)
```

ID/Feature0	1	2	3	4	5	...	53
12854	416948	0	57260	18	0.0	1122.0	...	21
17285	683375	21076	16319	31	0.0	419.0	...	3
18898	55275	0	1834	13	0.0	411.0	...	2
30197	1959278	0	12641	18	0.0	513.0	...	0
38069	83215	393	12259	18	0.0	73.0	...	0

Check the size of our dataset

```
print("Dataset has {} entries and {} features".format(*df.shape))
```

```
Dataset has 40949 entries and 54 features
```

In order to evaluate the performance of our model, we need to train it on a

sample of the data and test it on an other. We can do this easily with the function `train_test_split` from scikit-learn. First, let's extract the features and the target from our dataset.

```
X, y = df.loc[:,:52].values, df.loc[:,53].values
```

We keep 90% of the dataset for training, and 10% (or a `.1` part) for testing.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=.1,
random_state=42)
```

Loading data into DMatrices

As mentioned before, in order to use the native API for XGBoost, we will first need to build DMatrices.

```
import xgboost as xgb

dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)
```

Building a baseline model

We are going to use mean absolute error (MAE) to evaluate the quality of our predictions. MAE is a common and simple metric that has the advantage of being in the same unit as our target, which means it can be compared to target values and easily interpreted. You can compute MAE by summing the absolute errors between your predictions and the true values of the target and averaging over all observations, which can be written:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

You can read more about it here. MAE is easy to compute and scikit-learn provides a function to do just that!

```
from sklearn.metrics import mean_absolute_error
```

In order to get an idea of the range of MAE we would like to achieve here, we are going to build a baseline model, and save its score for later. This score is what we can achieve with no efforts, so we hope we will beat it with our fancy algorithm.

For our baseline, we will keep things simple and predict that each new post will get the mean number of comments that we observed in the training set.

```
import numpy as np
# "Learn" the mean from the training data
mean_train = np.mean(y_train)

# Get predictions on the test set
baseline_predictions = np.ones(y_test.shape) * mean_train

# Compute MAE
mae_baseline = mean_absolute_error(y_test, baseline_predictions)

print("Baseline MAE is {:.2f}".format(mae_baseline))

Baseline MAE is 11.31
```

That is, the prediction is, on average, 11.31 comments off from the actual number of comments a post receives. Is that good? Well not really, if you look at the mean we just computed, you will see that the average number of comments for a post in the training set is a bit more than 7. Our baseline

is, on average, off by more comments than that…

Training and Tuning an XGBoost model

Quick note on the method

In the following, we are going to see methods to tune the main parameters of your XGBoost model. In an ideal world, with infinite resources and where time is not an issue, you could run a giant grid search with all the parameters together and find the optimal solution.

In fact, you might even be able to do that with really small datasets, but as the data grows bigger, training time grows too, and each step in the tuning process becomes more expensive. For this reason it is important to understand the role of the parameters and focus on the steps that we expect to impact our results the most. Here we will tune 6 of the hyperparameters that are usually having a big impact on performance. Whilst, again, it would be necessary to test all combinations to ensure we find *THE* optimal solution, our goal here is to find a good enough one by improving our out-of-the-box model with as few steps as possible.

The params dictionary

Most of the parameters passed to XGBoost via the native API are defined in a dictionary. Let's define it with default values for the moment. You can find a list and a description of all parameters here

```
params = {
    # Parameters that we are going to tune.
    'max_depth':6,
    'min_child_weight': 1,
    'eta':.3,
    'subsample': 1,
    'colsample_bytree': 1,
    # Other parameters
    'objective':'reg:linear',
}
```

Parameters `num_boost_round` and `early_stopping_rounds`

The first parameter we will look at is not part of the `params` dictionary, but will be passed as a standalone argument to the training method. This parameter is called `num_boost_round` and corresponds to the number of boosting rounds or trees to build. Its optimal value highly depends on the other parameters, and thus it should be re-tuned each time you update a parameter. You could do this by tuning it together with all parameters in a grid-search, but it requires a lot of computational effort.

Fortunately XGBoost provides a nice way to find the best number of rounds whilst training. Since trees are built sequentially, instead of fixing the number of rounds at the beginning, we can test our model at each step and see if adding a new tree/round improves performance.

To do so, we define a test dataset and a metric that is used to assess performance at each round. If performance haven't improved for `N` rounds (`N` is defined by the variable `early_stopping_round`), we stop the training and keep the best number of boosting rounds. Let's see how to use it.

First, we need to add the evaluation metric we are interested in to our `params` dictionary.

```
params['eval_metric'] = "mae"
```

We still need to pass a `num_boost_round` which corresponds to the maximum number of boosting rounds that we allow. We set it to a large value hoping to find the optimal number of rounds before reaching it, if we haven't improved performance on our test dataset in `early_stopping_round` rounds

```
num_boost_round = 999
```

In order to automatically find the best number of boosting rounds, we need to pass extra parameters on top of the `params` dictionary, the training DMatrix and `num_boost_round`:

- `evals`: a list of pairs `(test_dmatrix, name_of_test)`. Here we will use our dtest DMatrix.
- `early_stopping_rounds`: The number of rounds without improvements after which we should stop, here we set it to `10`.

```
model = xgb.train(
    params,
    dtrain,
    num_boost_round=num_boost_round,
    evals=[(dtest, "Test")],
    early_stopping_rounds=10
)
```

```
[0] Test-mae:5.97478
Will train until Test-mae hasn't improved in 10 rounds.
[1] Test-mae:5.03359
…
[6] Test-mae:4.31315
[7] Test-mae:4.33087
…
[15]    Test-mae:4.39104
[16]    Test-mae:4.40307
Stopping. Best iteration:
[6] Test-mae:4.31315
```

```
print("Best MAE: {:.2f} with {} rounds".format(
                 model.best_score,
                 model.best_iteration+1))
```

```
Best MAE: 4.31 with 7 rounds
```

As you can see we stopped before reaching the maximum number of boosting rounds, that's because after the 7th tree, adding more rounds did not lead to improvements of MAE on the test dataset.

Let's keep this MAE in mind for later, this is the MAE of our model with default parameters and an optimal number of boosting rounds, on the test dataset. As you can see, we are already beating the baseline.

Using XGBoost's CV

In order to tune the other hyperparameters, we will use the `cv` function from XGBoost. It allows us to run cross-validation on our training dataset and returns a mean MAE score.

We need to pass it:

- `params` : our dictionary of parameters.
- our `dtrain` matrix.
- `num_boost_round` : number of boosting rounds. Here we will use a large number again and count on `early_stopping_rounds` to find the optimal number of rounds before reaching the maximum.
- `seed` : random seed. It's important to set a seed here, to ensure we are using the same folds for each step so we can properly compare the scores with different parameters.
- `nfold` : the number of folds to use for cross-validation
- `metrics` : the metrics to use to evaluate our model, here we use MAE.

As you can see, we don't need to pass a test dataset here. It's because the cross-validation function is splitting the train dataset into `nfolds` and iteratively keeps one of the folds for test purposes. You can read more about it here.

Let's see what cross-validation score we get with our current parameters:

```
cv_results = xgb.cv(
    params,
    dtrain,
    num_boost_round=num_boost_round,
```

```
    seed=42,
    nfold=5,
    metrics={'mae'},
    early_stopping_rounds=10
)

cv_results
```

Round	test-mae-mean	test-mae-std	train-mae-mean	train-mae-std
0	5.690	0.270	5.605	0.0646
1	4.850	0.272	4.623	0.0651
2	4.469	0.239	4.060	0.0658
3	4.269	0.224	3.723	0.0607
4	4.193	0.190	3.511	0.0611
5	4.173	0.189	3.368	0.0610
etc.	etc.	etc.	etc.	etc.
36	4.110	0.216	2.287	0.0384

`cv` returns a table where the rows correspond to the number of boosting trees used, here again, we stopped before the 999 rounds (fortunately!).

The 4 columns correspond to the mean and standard deviation of MAE on the test dataset and on the train dataset. For this tutorial we will only try to improve the mean test MAE. We can get the best MAE score from `cv` with:

```
cv_results['test-mae-mean'].min()

4.1095786000000007
```

Now that we know how to use `cv`, we are ready to start tuning! We will first tune our parameters to minimize the MAE on cross-validation, and then check the performance of our model on the test dataset.

Parameters `max_depth` and `min_child_weight`

Those parameters add constraints on the architecture of the trees.

- `max_depth` is the maximum number of nodes allowed from the root to the farthest leaf of a tree. Deeper trees can model more complex relationships by adding more nodes, but as we go deeper, splits become less relevant and are sometimes only due to noise, causing the model to overfit.
- `min_child_weight` is the minimum weight (or number of samples if all samples have a weight of 1) required in order to create a new node in the tree. A smaller `min_child_weight` allows the algorithm to create children that correspond to fewer samples, thus allowing for more complex trees, but again, more likely to overfit.

Thus, those parameters can be used to control the complexity of the trees. It is important to tune them together in order to find a good trade-off between model bias and variance

Let's make a list containing all the combinations `max_depth` / `min_child_weight` that we want to try.

```
# You can try wider intervals with a larger step between
# each value and then narrow it down. Here after several
# iteration I found that the optimal value was in the
# following ranges.
gridsearch_params = [
    (max_depth, min_child_weight)
    for max_depth in range(9,12)
    for min_child_weight in range(5,8)
]
```

Let's run cross validation on each of those pairs. It can take some time…

```
# Define initial best params and MAE
min_mae = float("Inf")
best_params = None
for max_depth, min_child_weight in gridsearch_params:
    print("CV with max_depth={}, min_child_weight={}".format(
                             max_depth,
                             min_child_weight))

    # Update our parameters
    params['max_depth'] = max_depth
```

```
        params['min_child_weight'] = min_child_weight

        # Run CV
        cv_results = xgb.cv(
            params,
            dtrain,
            num_boost_round=num_boost_round,
            seed=42,
            nfold=5,
            metrics={'mae'},
            early_stopping_rounds=10
        )

        # Update best MAE
        mean_mae = cv_results['test-mae-mean'].min()
        boost_rounds = cv_results['test-mae-mean'].argmin()
        print("\tMAE {} for {} rounds".format(mean_mae, boost_rounds))
        if mean_mae < min_mae:
            min_mae = mean_mae
            best_params = (max_depth,min_child_weight)

print("Best params: {}, {}, MAE: {}".format(best_params[0],
best_params[1], min_mae))
```

```
CV with max_depth=9, min_child_weight=5
    MAE 4.0445676 for 6 rounds
CV with max_depth=9, min_child_weight=6
    MAE 4.0772509999999995 for 5 rounds
CV with max_depth=9, min_child_weight=7
    MAE 4.059255 for 5 rounds
CV with max_depth=10, min_child_weight=5
    MAE 4.088694599999999 for 5 rounds
CV with max_depth=10, min_child_weight=6
    MAE 4.0365786 for 5 rounds
CV with max_depth=10, min_child_weight=7
    MAE 4.0846622 for 5 rounds
CV with max_depth=11, min_child_weight=5
    MAE 4.0630098 for 5 rounds
CV with max_depth=11, min_child_weight=6
    MAE 4.0564924 for 5 rounds
CV with max_depth=11, min_child_weight=7
    MAE 4.064484800000001 for 5 rounds
Best params: 10, 6, MAE: 4.0365786
```

We get the best score with a `max_depth` of 10 and `min_child_weight` of 6, so let's update our params

```
params['max_depth'] = 10
params['min_child_weight'] = 6
```

Parameters `subsample` and `colsample_bytree`

Those parameters control the sampling of the dataset that is done at each boosting round.

Instead of using the whole training set every time, we can build a tree on slightly different data at each step, which makes it less likely to overfit to a single sample or feature.

- `subsample` corresponds to the fraction of observations (the rows) to subsample at each step. By default it is set to 1 meaning that we use all rows.
- `colsample_bytree` corresponds to the fraction of features (the columns) to use. By default it is set to 1 meaning that we will use all features.

Let's see if we can get better results by tuning those parameters together.

```
gridsearch_params = [
    (subsample, colsample)
    for subsample in [i/10. for i in range(7,11)]
    for colsample in [i/10. for i in range(7,11)]
]
```

This can take some time…

```
min_mae = float("Inf")
best_params = None

# We start by the largest values and go down to the smallest
for subsample, colsample in reversed(gridsearch_params):
    print("CV with subsample={}, colsample={}".format(
                             subsample,
                             colsample))

    # We update our parameters
    params['subsample'] = subsample
    params['colsample_bytree'] = colsample

    # Run CV
    cv_results = xgb.cv(
        params,
        dtrain,
        num_boost_round=num_boost_round,
        seed=42,
```

```
            nfold=5,
            metrics={'mae'},
            early_stopping_rounds=10
        )

        # Update best score
        mean_mae = cv_results['test-mae-mean'].min()
        boost_rounds = cv_results['test-mae-mean'].argmin()
        print("\tMAE {} for {} rounds".format(mean_mae, boost_rounds))
        if mean_mae < min_mae:
            min_mae = mean_mae
            best_params = (subsample,colsample)

print("Best params: {}, {}, MAE: {}".format(best_params[0],
best_params[1], min_mae))
```

```
CV with subsample=1.0, colsample=1.0
    MAE 4.0365786 for 5 rounds
CV with subsample=1.0, colsample=0.9
    MAE 4.0535088 for 5 rounds
CV with subsample=1.0, colsample=0.8
    MAE 4.0725374 for 5 rounds
CV with subsample=1.0, colsample=0.7
    MAE 4.1336508 for 5 rounds
CV with subsample=0.9, colsample=1.0
    MAE 4.0891608 for 4 rounds
CV with subsample=0.9, colsample=0.9
    MAE 4.123717600000001 for 6 rounds
CV with subsample=0.7, colsample=0.7
…
    MAE 4.3136622000000004 for 7 rounds
Best params: 0.8, 1.0, MAE: 4.0223654
```

Again, we update our `params` dictionary.

```
params['subsample'] = .8
params['colsample_bytree'] = 1.
```

Parameter `ETA`

The `ETA` parameter controls the learning rate. It corresponds to the shrinkage of the weights associated to features after each round, in other words it defines the amount of "correction" we make at each step (remember how each boosting round is correcting the errors of the previous? if not, check our first tutorial here).

In practice, having a lower `eta` makes our model more robust to

overfitting thus, usually, the lower the learning rate, the best. But with a lower `eta`, we need more boosting rounds, which takes more time to train, sometimes for only marginal improvements. Let's try a couple of values here, and time them with the notebook command:

```
%time

# This can take some time…
min_mae = float("Inf")
best_params = None

for eta in [.3, .2, .1, .05, .01, .005]:
    print("CV with eta={}".format(eta))

    # We update our parameters
    params['eta'] = eta

    # Run and time CV
    %time cv_results = xgb.cv(
            params,
            dtrain,
            num_boost_round=num_boost_round,
            seed=42,
            nfold=5,
            metrics=['mae'],
            early_stopping_rounds=10
          )

    # Update best score
    mean_mae = cv_results['test-mae-mean'].min()
    boost_rounds = cv_results['test-mae-mean'].argmin()
    print("\tMAE {} for {} rounds\n".format(mean_mae, boost_rounds))
    if mean_mae < min_mae:
        min_mae = mean_mae
        best_params = eta

print("Best params: {}, MAE: {}".format(best_params, min_mae))
```

```
CV with eta=0.3
CPU times: user 22.3 s, sys: 72 ms, total: 22.4 s
Wall time: 3.04 s
    MAE 4.0223654 for 6 rounds

CV with eta=0.2
CPU times: user 26.4 s, sys: 88 ms, total: 26.5 s
Wall time: 3.55 s
    MAE 3.9568890000000003 for 9 rounds

…

Run CV with eta=0.01
CPU times: user 5min 22s, sys: 332 ms, total: 5min 23s
Wall time: 42.1 s
```

```
    MAE 3.8394792000000004 for 247 boosting rounds

Run CV with eta=0.005
CPU times: user 10min 11s, sys: 620 ms, total: 10min 12s
Wall time: 1min 19s
    MAE 3.8305794000000004 for 463 rounds

Best params: 0.005, MAE: 3.8305794000000004
```

As you can see with the 2 last steps, by reducing `eta` from .01 to .005 we saved only ~.009 in MAE but went from 44s to 1min19s. It looks like we start converging and our MAE is not getting much better. Depending on your goal, you might want to take the extra time for the little improvement in MAE, but here we'll stick to .01.

```
params['eta'] = .01
```

Results

Here is how our final dictionary of parameters looks like:

```
params
```

```
{'colsample_bytree': 1.0,
 'eta': 0.01,
 'eval_metric': 'mae',
 'max_depth': 10,
 'min_child_weight': 6,
 'objective': 'reg:linear',
 'subsample': 0.8}
```

Let’s train a model with it and see how well it does on our test set!

```
model = xgb.train(
    params,
    dtrain,
    num_boost_round=num_boost_round,
    evals=[(dtest, "Test")],
    early_stopping_rounds=10
)
```

```
[0] Test-mae:7.69075
Will train until Test-mae hasn't improved in 10 rounds.
[1] Test-mae:7.62033
[2] Test-mae:7.55244
[3] Test-mae:7.48772
…
[205]   Test-mae:3.90244
[206]   Test-mae:3.90458
Stopping. Best iteration:
[196]   Test-mae:3.90198
```

```
print("Best MAE: {:.2f} in {} rounds".format(model.best_score,
model.best_iteration+1))
```

```
Best MAE: 3.90 in 197 rounds
```

As expected it took us more rounds to get there, but we improved our MAE from 4.31 to 3.90. Is that good? Well it depends what you compare it to. Noting that we got this improvement almost for free, without adding data or engineering features, simply by spending a bit of time tuning our model, then it's not bad. But it's good to notice that it did not transform a poor model (we are still off by 4 comments on average whilst our average number of comments is 7…) into an excellent one. This is quite common with Machine Learning, whilst it is important to "roughly" tune your model to get good results from it, it will only get you that far. And there is a point after which additional time spent tuning it only provides marginal improvements. When it's the case, it's usually worth looking more closely at the data to find better ways of extracting information, and/or try other algorithms instead of fine tuning your current model.

Saving your model

Although we found the best number of rounds, our model has been trained with more rounds than optimal, thus before using it for predictions, we should retrain it with the good number of rounds. Since we now the exact best `num_boost_round`, we don't need the `early_stopping_round` anymore.

```
num_boost_round = model.best_iteration + 1
```

```
best_model = xgb.train(
    params,
    dtrain,
```

```
    num_boost_round=num_boost_round,
    evals=[(dtest, "Test")]
)
```

```
[0] Test-mae:7.69075
[1] Test-mae:7.62033
[2] Test-mae:7.55244
…
[195]   Test-mae:3.90328
[196]   Test-mae:3.90198
```

All good, now let's use our model to make predictions. We will use the test dataset and compute MAE with the scikit-learn function. We should obtain the same score as promised in the last round of training, let's check!

```
mean_absolute_error(best_model.predict(dtest), y_test)
```

```
3.9019752631912303
```

Great! If you want to re-use your model on new data in the future, it can be a good idea to save it to a file, here is how you can do it with XGBoost:

```
best_model.save_model("my_model.model")
```

You can then load the model later with:

About the author: Kevin Lemagnen

```
loaded_model = xgb.Booster()
loaded_model.load_model("my_model.model")
# And use it for predictions.
loaded_model.predict(dtest)
array([4.1771087 , 3.3485281 , 1.9063734 , ..., 3.1688757 ,
       0.1173397 ,  3.41118741], dtype=float32)
```

Kevin is a Data Scientist working at a startup based in Cambridge that is using machine learning to solve problems for the energy industry. Prior to that he worked for 2 years at a large technology company, focusing on the Internet of Things. Kevin studied Nanotechnologies for his MSc and Physics, Electronics and Telecommunications for his BSc.

If you are interested more about Spark and big data systems, check out our upcoming webinar series:

Thanks for reading! If you enjoyed the post, we'd appreciate your

support by applauding via the clap (👏) button below or by sharing this article so others can find it.

Data Science Webinar Series

Join us across the world to learn about Data Science, Big Data Analytics and state-of-the-art techniques used in...

cambridgespark.com

If you'd like to be the first to hear about our new content, including new tutorials, case studies and other useful resources, **click here**!

Data Science Big Data Analytics Xgboost Python Tutorial