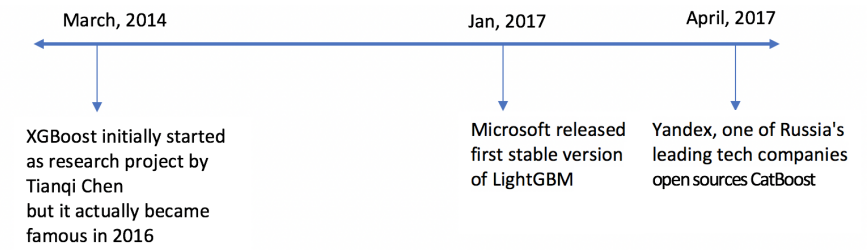
CatBoost vs. Light GBM vs. XGBoost

Alvira Swalin Follow
Mar 13, 2018 · 8 min read ★

Who is going to win this war of predictions and on what cost? Let's explore.

I recently participated in this Kaggle competition (WIDS Datathon by Stanford) where I was able to land up in Top 10 using various boosting algorithms. Since then, I have been very curious about the fine workings of each model including parameter tuning, pros and cons and hence decided to write this blog. Despite the recent re-emergence and popularity of neural networks, I am focusing on boosting algorithms because they are still more useful in the regime of limited training data, little training time and little expertise for parameter tuning.



Since XGBoost (often called GBM Killer) has been in the machine learning world for a longer time now with lots of articles dedicated to it, this post will focus more on CatBoost & LGBM. Below are the topics we will cover-

- Structural Differences
- Treatment of categorical variables by each algorithm
- Understanding Parameters
- Implementation on Dataset
- Performance of each algorithm

Structural Differences in LightGBM & XGBoost

LightGBM uses a novel technique of Gradient-based One-Side Sampling (GOSS) to filter out the data instances for finding a split value while XGBoost uses pre-sorted algorithm & Histogram-based algorithm for computing the best split. Here instances mean observations/samples.

First, let us understand how pre-sorting splitting works-

- For each node, enumerate over all features
- For each feature, sort the instances by feature value
- Use a linear scan to decide the best split along that feature basis information gain
- Take the best split solution along all the features

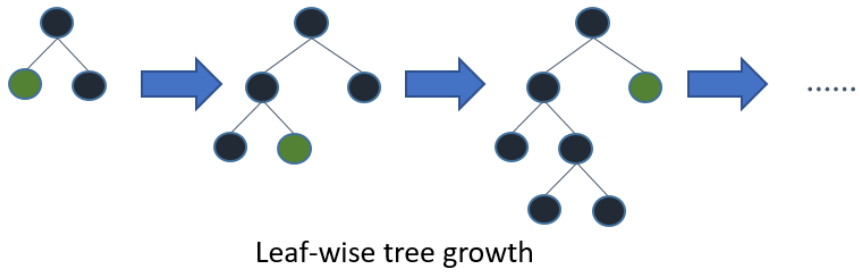
In simple terms, Histogram-based algorithm splits all the data points for a feature into discrete bins and uses these bins to find the split value of histogram. While, it is efficient than pre-sorted algorithm in training speed which enumerates all possible split points on the pre-sorted feature values, it is still behind GOSS in terms of speed.

So what makes this GOSS method efficient?
In AdaBoost, the sample weight serves as a good indicator for the importance of samples. However, in Gradient Boosting Decision Tree (GBDT), there are no native sample weights, and thus the sampling methods proposed for AdaBoost cannot be directly applied. Here comes gradient-based sampling.

> *Gradient represents the slope of the tangent of the loss function, so logically if gradient of data points are large in some sense, these points are important for finding the optimal split point as they have higher error*

GOSS keeps all the instances with large gradients and performs random sampling on the instances with small gradients. For example, let's say I have 500K rows of data where 10k rows have higher gradients. So my algorithm will choose (10k rows of higher gradient+ x% of remaining 490k rows chosen randomly). Assuming x is 10%, total rows selected are 59k out of 500K on the basis of which split value if found.

> *The basic assumption taken here is that samples with training instances with small gradients have smaller training error and it is already well-trained.*
> *In order to keep the same data distribution, when computing the information gain, GOSS introduces a constant multiplier for the data instances with small gradients. Thus, GOSS achieves a good balance between reducing the number of data instances and keeping the accuracy for learned decision trees.*



Leaf with higher gradient/error is used for growing further in LGBM

How each model treats Categorical Variables?

CatBoost

CatBoost has the flexibility of giving indices of categorical columns so that it can be encoded as one-hot encoding using one_hot_max_size (Use one-hot encoding for all features with number of different values less than or equal to the given parameter value).

If you don't pass any anything in cat_features argument, CatBoost will treat all the columns as numerical variables.

> ***Note: If a column having string values is not provided in the cat_features, CatBoost throws an error. Also, a column having default int type will be treated as numeric by default, one has to specify it in cat_features to make the algorithm treat it as categorical.***

```
from catboost import CatBoostRegressor
# Initialize data
cat_features = [0,1,2]
train_data = [["a","b",1,4,5,6],["a","b",4,5,6,7],["c","d",30,40,50,60]]
test_data = [["a","b",2,4,6,8],["a","d",1,4,50,60]]
train_labels = [10,20,30]
# Initialize CatBoostRegressor
model = CatBoostRegressor(iterations=2, learning_rate=1, depth=2)
# Fit model
model.fit(train_data, train_labels, cat_features)
```

For remaining categorical columns which have unique number of categories greater than one_hot_max_size, CatBoost uses an efficient method of encoding which is similar to mean encoding but reduces overfitting. The process goes like this—
1. Permuting the set of input observations in a random order. Multiple random permutations are generated
2. Converting the label value from a floating point or category to an integer
3. All categorical feature values are transformed to numeric values using the following formula:

$$avg\_target = \frac{countInClass + prior}{totalCount + 1}$$

Where, **CountInClass** is how many times the label value was equal to "1" for objects with the current categorical feature value
Prior is the preliminary value for the numerator. It is determined by the starting parameters. **TotalCount** is the total number of objects (up to the current one) that have a categorical feature value matching the current one.
Mathematically, this can be represented using below equation:

$$\text{Let } \sigma = (\sigma_1, \ldots, \sigma_n) \text{ be the permutation, then } x_{\sigma_p,k} \text{ is substituted with } \frac{\sum_{j=1}^{p-1} [x_{\sigma_j,k} = x_{\sigma_p,k}] Y_{\sigma_j} + a \cdot P}{\sum_{j=1}^{p-1} [x_{\sigma_j,k} = x_{\sigma_p,k}] + a}, \quad (1)$$

LightGBM

Similar to CatBoost, LightGBM can also handle categorical features by taking the input of feature names. It does not convert to one-hot coding, and is much faster than one-hot coding. LGBM uses a special algorithm to find the split value of categorical features [Link].

Specific feature names and categorical features:

```
train_data = lgb.Dataset(data, label=label, feature_name=['c1', 'c2', 'c3'],
↪categorical_feature=['c3'])
```

> ***Note: You should convert your categorical features to int type before you construct Dataset for LGBM. It does not accept string values even if you passes it through categorical_feature parameter.***

XGBoost

Unlike CatBoost or LGBM, XGBoost cannot handle categorical features by itself, it only accepts numerical values similar to Random Forest. Therefore one has to perform various encodings like label encoding, mean encoding or one-hot encoding before supplying categorical data to XGBoost.

Similarity in Hyperparameters

All these models have lots of parameters to tune but we will cover only the important ones. Below is the list of these parameters according to their function and their counterparts across different models.

Function	XGBoost	CatBoost	Light GBM
Important parameters which control overfitting	1. **learning_rate or eta** – optimal values lie between 0.01-0.2 2. **max_depth** 3. **min_child_weight:** similar to min_child leaf; default is 1	1. **Learning_rate** 2. **Depth** - value can be any integer up to 16. Recommended - [1 to 10] 3. No such feature like min_child_weight 4. **l2-leaf-reg**: L2 regularization coefficient. Used for leaf value calculation (any positive integer allowed)	1. **learning_rate** 2. **max_depth**: default is 20. Important to note that tree still grows leaf-wise. Hence it is important to tune **num_leaves** (number of leaves in a tree) which should be smaller than 2^(max_depth). It is a very important parameter for LGBM 3. **min_data_in_leaf**: default=20, alias= min_data, min_child_samples
Parameters for categorical values	Not Available	1. **cat_features**: It denotes the index of categorical features 2. **one_hot_max_size**: Use one-hot encoding for all features with number of different values less than or equal to the given parameter value (max – 255)	1. **categorical_feature:** specify the categorical features we want to use for training our model
Parameters for controlling speed	1. **colsample_bytree**: subsample ratio of columns 2. **subsample**: subsample ratio of the training instance 3. **n_estimators**: maximum number of decision trees; high value can lead to overfitting	1. **rsm:** Random subspace method. The percentage of features to use at each split selection 2. No such parameter to subset data 3. **iterations**: maximum number of trees that can be built; high value can lead to overfitting	1. **feature_fraction:** fraction of features to be taken for each iteration 2. **bagging_fraction:** data to be used for each iteration and is generally used to speed up the training and avoid overfitting 3. **num_iterations:** number of boosting iterations to be performed; default=100

Implementation on a Dataset

I am using the Kaggle Dataset of flight delays for the year 2015 as it has both categorical and numerical features. With approximately 5 million rows, this dataset will be good for judging the performance in terms of both speed and accuracy of tuned models for each type of boosting. I will be using a 10% subset of this data ~ 500k rows.
Below are the features used for modeling:

- **MONTH, DAY, DAY_OF_WEEK**: data type int
- **AIRLINE and FLIGHT_NUMBER**: data type int
- **ORIGIN_AIRPORT** and **DESTINATION_AIRPORT:** data type string
- **DEPARTURE_TIME:** data type float

- **ARRIVAL_DELAY**: this will be the target and is transformed into boolean variable indicating delay of more than 10 minutes
- **DISTANCE and AIR_TIME**: data type float

```
import pandas as pd, numpy as np, time
from sklearn.model_selection import train_test_split

data = pd.read_csv("flights.csv")
data = data.sample(frac = 0.1, random_state=10)

data = data[["MONTH","DAY","DAY_OF_WEEK","AIRLINE","FLIGHT_
                 "ORIGIN_AIRPORT","AIR_TIME", "DEPARTURE_TI
data.dropna(inplace=True)

data["ARRIVAL_DELAY"] = (data["ARRIVAL_DELAY"]>10)*1

cols = ["AIRLINE","FLIGHT NUMBER","DESTINATION AIRPORT","OR
```

XGBoost

```
import xgboost as xgb
from sklearn import metrics

def auc(m, train, test):
    return (metrics.roc_auc_score(y_train,m.predict_proba(t
                            metrics.roc_auc_score(y_test,m.

# Parameter Tuning
model = xgb.XGBClassifier()
param_dist = {"max_depth": [10,30,50],
              "min_child_weight" : [1,3,6],
              "n_estimators": [200],
              "learning_rate": [0.05, 0.1,0.16],}
grid_search = GridSearchCV(model, param_grid=param_dist, cv
                                    verbose=10, n_jobs=-1)
grid_search.fit(train, y_train)
```

Light GBM

```
import lightgbm as lgb
from sklearn import metrics

def auc2(m, train, test):
    return (metrics.roc_auc_score(y_train,m.predict(train))
                            metrics.roc_auc_score(y_test,m.

lg = lgb.LGBMClassifier(silent=False)
param_dist = {"max_depth": [25,50, 75],
              "learning_rate" : [0.01,0.05,0.1],
              "num_leaves": [300,900,1200],
              "n_estimators": [200]
             }
grid_search = GridSearchCV(lg, n_jobs=-1, param_grid=param_
grid_search.fit(train,y_train)
grid_search.best_estimator_

d_train = lgb.Dataset(train, label=y_train)
params = {"max_depth": 50, "learning_rate" : 0.1, "num_leav

```

CatBoost

While tuning parameters for CatBoost, it is difficult to pass indices for categorical features. Therefore, I have tuned parameters without passing categorical features and evaluated two model—one with and other without categorical features. I have separately tuned one_hot_max_size because it does not impact the other parameters.

```
import catboost as cb
cat_features_index = [0,1,2,3,4,5,6]

def auc(m, train, test):
    return (metrics.roc_auc_score(y_train,m.predict_proba(t
                            metrics.roc_auc_score(y_test,m.

params = {'depth': [4, 7, 10],
          'learning_rate' : [0.03, 0.1, 0.15],
         'l2_leaf_reg': [1,4,9],
         'iterations': [300]}
cb = cb.CatBoostClassifier()
cb_model = GridSearchCV(cb, params, scoring="roc_auc", cv =
cb_model.fit(train, y_train)

With Categorical features
clf = cb.CatBoostClassifier(eval_metric="AUC", depth=10, it
```

Results

	XGBoost	Light BGM		CatBoost	
Parameters Used	max_depth: 50 learning_rate: 0.16 min_child_weight: 1 n_estimators: 200	max_depth: 50 learning_rate: 0.1 num_leaves: 900 n_estimators: 300		depth: 10 learning_rate: 0.15 l2_leaf_reg= 9 iterations: 500 one_hot_max_size = 50	
Training AUC Score	0.999	Without passing indices of categorical features	Passing indices of categorical features	Without passing indices of categorical features	Passing indices of categorical features
		0.992	0.999	0.842	0.887
Test AUC Score	0.789	0.785	0.772	0.752	0.816
Training Time	970 secs	153 secs	326 secs	180 secs	390 secs
Prediction Time	184 secs	40 secs	156 secs	2 secs	14 secs
Parameter Tuning Time (for 81 fits, 200 iteration)	500 minutes	200 minutes		120 minutes	

End Notes

For evaluating model, we should look into the performance of model in terms of both speed and accuracy.

Keeping that in mind, CatBoost comes out as the winner with maximum accuracy on test set (0.816), minimum overfitting (both train and test accuracy are close) and minimum prediction time & tuning time. But this happened only because we considered categorical variables and tuned one_hot_max_size. If we don't take advantage of these features of CatBoost, it turned out to be the worst performer with just 0.752 accuracy. Hence we learnt that CatBoost performs well only when we have categorical variables in the data and we properly tune them.

Our next performer was XGBoost which generally works well. It's accuracy was quite close to CatBoost even after ignoring the fact that we have categorical variables in the data which we had converted into numerical values for its consumption. However, the only problem with XGBoost is that it is too slow. It was really frustrating to tune its parameters especially (took me 6 hours to run GridSearchCV—very bad idea!). The better way is to tune parameters separately rather than using GridSearchCV. Check out this blog post to understand how to tune parameters smartly.

Finally, the last place goes to Light GBM. An important thing to note here is that it performed poorly in terms of both speed and accuracy when cat_features is used. I believe the reason why it performed badly was because it uses some kind of modified mean encoding for categorical data which caused overfitting (train accuracy is quite high —0.999 compared to test accuracy). However if we use it normally like XGBoost, it can achieve similar (if not higher) accuracy with much faster speed compared to XGBoost
(LGBM—0.785, XGBoost—0.789).

Lastly, I have to say that these observations are true for this particular dataset and may or may not remain valid for other datasets. However, one thing which is true in general is that XGBoost is slower than the other two algorithms.

So which one is your favorite? Please comment with the reasons.
Any feedback or suggestions for improvement will be really appreciated!

Check out my other blogs here!

LinkedIn: www.linkedin.com/in/alvira-swalin

Resources

1. http://learningsys.org/nips17/assets/papers/paper_11.pdf
2. https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf
3. https://arxiv.org/pdf/1603.02754.pdf
4. https://github.com/Microsoft/LightGBM
5. https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/
6. https://stats.stackexchange.com/questions/307555/mathematical-differences-between-gbm-xgboost-lightgbm-catboost