Erciyes Üniversitesi
Fen Bilimleri Enstitüsü Dergisi
Cilt 37, Sayı 1 , 2021

Erciyes University
Journal of Institue Of Science and Technology
Volume 37, Issue 1, 2021

Comparison of Gradient Boosting Decision Tree Algorithms for CPU Performance

Haithm ALSHARI[*1], Abdulrazak Yahya SALEH[2], Alper ODABAŞ[3]

[*1] Eskişehir Osmangazi Üniversitesi Matematik ve Bilgisayar Bilimleri Bölümü, ESKIŞEHIR
[2] FSKPM Faculty, Malaysia Sarawak University (UNIMAS), 94300 Kota Samarahan, SARAWAK
[3] Eskişehir Osmangazi Üniversitesi Matematik ve Bilgisayar Bilimleri Bölümü, ESKIŞEHIR

(Alınış / Received: 14.02.2021, Kabul / Accepted: 15.04.2021, Online Yayınlanma / Published Online: 28.04.2021)

Keywords
Decision Tree,
Gradient Boosting,
XGBoost,
LightGBM,
CatBoost

Abstract: Gradient Boosting Decision Trees (GBDT) algorithms have been proven to be among the best algorithms in machine learning. XGBoost, the most popular GBDT algorithm, has won many competitions on websites like Kaggle. However, XGBoost is not the only GBDT algorithm with state-of-the-art performance. There are other GBDT algorithms that have more advantages than XGBoost and sometimes even more potent like LightGBM and CatBoost. This paper aims to compare the performance of CPU implementation of the top three gradient boosting algorithms. We start by explaining how the three algorithms work and the hyperparameters similarities between them. Then we use a variety of performance criteria to evaluate their performance. We divide the performance criteria into four: accuracy, speed, reliability, and ease of use. The performance of the three algorithms has been tested with five classification and regression problems. Our findings show that the LightGBM algorithm has the best performance of the three with a balanced combination of accuracy, speed, reliability, and ease of use, followed by XGBoost with the histogram method, and CatBoost came last with slow and inconsistent performance.

CPU Performansı için Gradyan Artırıcı Karar Ağacı Algoritmalarının Karşılaştırılması

Anahtar Kelimeler
Karar Ağacı,
Gradyan Artırıcı,
XGBoost,
LightGBM,
CatBoost

Öz: Gradyan Artırıcı Karar Ağacı (GBDT) algoritmalarının regresyon ve sınıflandırma problemlerinin çüzümünde makine öğrenimindeki en iyi algoritmalar arasında olduğu kanıtlanmıştır. Kaggle gibi web sitelerinin düzenlediği birçok yarışmayı kazanması sebebiyle en popüler GBDT algoritması olan XGBoost son teknoloji performansa sahip tek GBDT algoritması değildir. LightGBM ve CatBoost gibi kimi zaman XGBoost'a göre daha fazla avantajları olan başka GBDT algoritmaları da vardır. Bu makale, en iyi üç gradyan artırıcı algoritmanın işlemci(CPU) performansını karşılaştırmayı amaçlamaktadır. Bunun için ilk olarak bu üç algoritmanın nasıl çalıştığını ve aralarındaki hiperparametre benzerliklerini açıklayacağız. Daha sonra performanslarını değerlendirmek için doğruluk, hız, güvenilirlik ve kullanım kolaylığı olarak dörde ayırdığımız performans kriterleri kullanacağız. Üç algoritmanın performansı beş sınıflandırma ve regresyon problemi ile test edilmiştir. Bulgularımız, LightGBM algoritmasının, dengeli bir doğruluk, hız, güvenilirlik ve kullanım kolaylığı kombinasyonuyla üçü arasında en iyi performansa sahip olduğunu, bunu histogram yöntemiyle XGBoost'un izlediğini ve CatBoost'un ise özellikle yavaş ve tutarsız performansla diğerlerinin gerisinde kaldığını göstermektedir.

***İlgili Yazar:** aodabas@ogu.edu.tr

1. Introduction

Decision trees are a machine learning method that uses a tree-like graph or model of decisions to split the dataset to either classifying or predicting based on features. A decision tree is a flow chart-like structure in which each internal node represents a test on a feature and each branch divides the data into one of two groups like in Figure 1. In the prediction process of the decision tree, the data is assigned to the suitable node, and the result of the nodes' test is the prediction of that node.

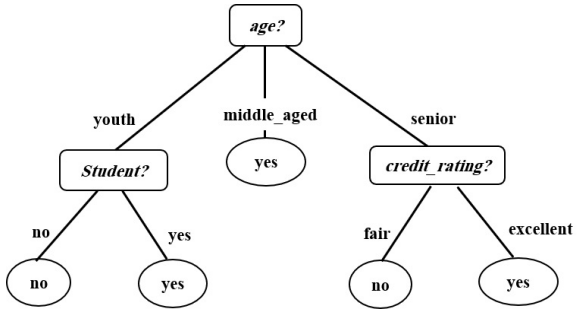

Figure 1. A simple graph shows a decision tree that is predicting whether a person is going to buy a PC or not. We can see that the model prediction shows that a young student will buy a computer.

Decision trees are known to be flexible and easy to interpret. However, using only one decision tree can lead to overfitting and this will affect the model generalization. We can use several ways to restrict decision tree's flexibility, for instance, limiting the tree depth, but using such methods will drive the decision tree toward underfit. This is the reason that we use multiple decisions tree combined together (ensemble) instead of a single decision tree. This method helps us make predictions that generalize well. There are many ways to combine multiple decision trees together; one of them is the gradient boosting decision tree.

Despite their strength, the core idea of GBTDs is very simple: first train several decision trees then taking their predictions and adding them together to get the final prediction. For instance, let assume that we try to predict health care prices; the predicted price of any health care service would be acquired by summing the predictions of all the decision trees.

GBDT uses iterative training, i.e., it builds trees one by one. For example, a GBDT that tries to predict health care costs would start by building a plain, weak decision tree based on the data with the raw health care costs. The goal in the training of decision tree is to minimize an objective function, for example, a loss function like Mean Absolute Error, through repetitively splitting the data in a way that maximizes some criterion until some limit – like trees' depth – is met. Figure 2 shows how recursive processing works when training a decision tree. The criterion is being selected in a way that minimizes the loss function in each split. Gini index is a commonly used criterion in binary classification, which measures the impurity of a node.

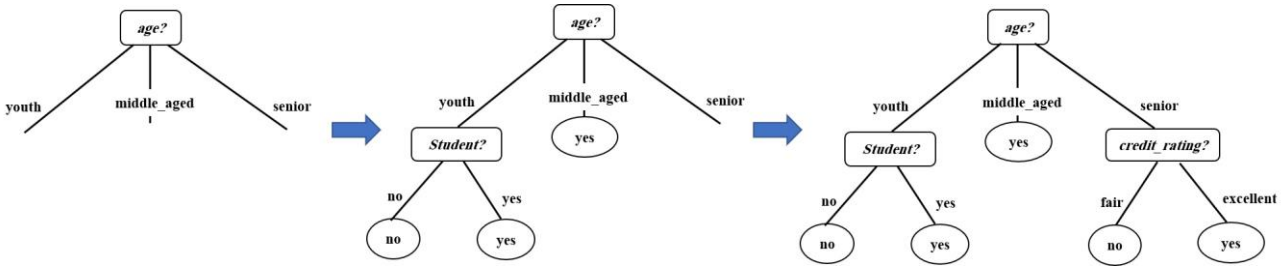

Figure 2. The recursive processing of training a decision tree.

Next, GBDT builds the next tree in a way that minimizes the loss function then takes the outputs of this tree and adds it to the first tree. And this can be achieved – approximately - by recursively splitting the data based on a new criterion. While the details of the criterion are complicated, it can be simple to calculate any criterion for any split of data by using the gradient statistics (the gradients' value for each data point) [1,2]

One thing to keep in mind that in order to compute the best split the model has to go through several splits as well as computing the criterion for each split. Analytically there is no solution to find the best split at each step. As we go forward, we will see that this problem is a key challenge in training GBDTs.

Although there are many GBDT implementations besides XGBoost, LightGBM, and CatBoost, these three algorithms are treated like the only GBDT representatives due to three main reasons:

- Easy-to-use implementations on the three major operating system Windows, macOS, and Linux.
- Open-source software libraries for many programming languages like C++, Python, and R.
- They are fast to train and produce accurate result.

Moreover, one thing to keep a note on that the GBDT implementation in sklearn [3] is much slower than XGBoost, LightGBM, and CatBoost. All of the three algorithms differ in the optimization's specifics. In this paper, we will explain how each of those algorithms improves GBDTs to make model training more accurate and more efficient.

1.1. Growing the Tree

During growing the decision tree both XGBoost and LightGBM use the leaf-wise growth strategy. But during the training process of each decision tree while LightGBM employs the same strategy XGBoost employs a different strategy level-wise. Although XGBoost originally used the level-wise strategy recently it has implemented the leaf-wise strategy but only for the histogram-based method, whereas LightGBM only has the leaf-wise strategy [4]. The level-wise strategy keeps the trees balanced while the leaf-wise strategy reduces the loss more by splitting the leaf that has the most loss [5,6] as shown in Figure 3.

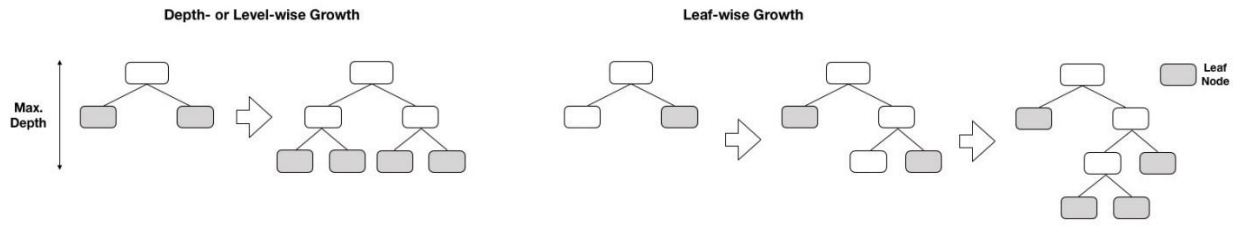

Figure 3. A graph shows how level-wise and leaf-wise growth strategies works.

Training with a level-wise strategy can be considered as a way of regularizing the training process. While any tree built with a level-wise strategy can be built with a leaf-wise strategy the opposite doesn't hold. Thus, while the training with a leaf-wise strategy is more flexible is also more prone to overfit. This the reason which makes the leaf-wise strategy a better choice when dealing with large datasets. Furthermore, in case of a leaf-wise growth strategy, a tree can grow deeper compared to a level-wise growth strategy while using the same number of leaves. This means while we use the same max_depth parameter on either of the strategies this will result in trees that have different levels of complexity [4]

In case of CatBoost, it uses symmetric or oblivious trees [7,8]. In each level of the tree, it uses the same features to split learning examples into the right and the left partitions which results in a tree that has a depth of k and precisely 2k leaves. Moreover, the leaf index can simply be calculated with bitwise operations. Therefore, the scheme of CatBoost learning can be considered depth-wise that has been simplified and obtained from a decision tree type. This type of tree brings many advantages in contrast to a classic one: More efficient GPU implementation, easy fitting scheme, capable of building really fast model appliers, tree structure acts as a regularization which can provide benefits for various tasks [9]. Building stages for a single tree in CatBoost [10].

First, the algorithm starts by doing the preliminary calculation of splits. Then, transforms the categorical features into numerical features (Optional). After that, it chooses the tree structure. And lastly, it calculates the values in leaves.

1.2. Finding the Best Split

The key challenge in GBDTs is to locate the best place to split the data for each leaf. When this process is naively done the algorithm will have to check every feature of every data point. Therefore, this will result in the computational complexity of O (n_data n_features) [6]. Nowadays we are dealing with datasets that have an enormous number of samples and features. For example, a dataset consists of 10 million documents and 1000 as a vocabulary size would have 10 billion entries. Thus, a naïve GBDT would take a very long time (sometimes forever for big data) to train on such a dataset. Although there isn't a method for finding the best split that doesn't

require checking all features of all dataset points, XGBoost, LightGBM and CatBoost present methods to find an approximation to of best split.

Exact Greedy Algorithm (XGBoost): XGBoost with level-wise strategy uses an exact greedy algorithm for computing the best split. In order to do so efficiently, the algorithm must first sort the data according to features [11]. We can summarize how pre-sorting splitting works in the following:

The algorithm enumerates over all the features for each node. Then, sorts the instances for each feature. Next, it decided the best split along feature's basis information gain by using a linear scan. Finally, it takes the fittest split solution throughout all the features.

Histogram-based method (XGBoost / LightGBM): The number of splits that the algorithm has to evaluate when building a tree directly affects its speed. Thus, fewer splits mean faster speed. Histogram-based methods employ the fact that often, introducing small changes to the split would not make much of a difference in the tree performance. Thus, the histogram-based method groups the features into a set of bins then use those bins instead of the feature for the splitting process as shown in Figure 4. When this method is used is like applying a subsample to the number of splits that will be evaluated by the model. Because when the algorithm binned the features before building each tree it can speed up training significantly and reduce the computational complexity to O (n_data n_bins). Despite the simplicity of the concept, histogram-based methods have more parameters to be chosen by the user.

1. Bins number which is a trade-off between speed and accuracy of the algorithm: when the user uses more bins the accuracy of the algorithm increases as well as the time it takes to complete the training.
2. The problem of which method to be used to divide the features into bins is not easy. If the user divides the bins using equal intervals (the simplest method) this can often lead to an unbalanced distribution of the data. In order to get the most balanced method for dividing the bins, we have to depend on the gradient statistics.

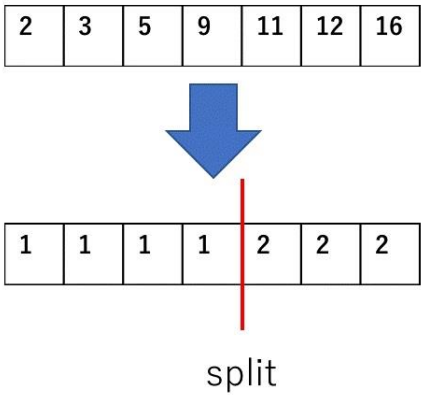

Figure 4. A graph shows how binning the feature reduces the number of splits to evaluate. But this method requires the features to be sorted in advance to be effective.

XGBoost algorithm offers the option tree_method=approx which uses the gradient statistics to compute a new set of bins at each split. Both LightGBM and XGBoost with histogram method compute the bins once at the beginning of training then use the same bins during the entire training process [6,12].

Gradient-based One-Side Sampling (LightGBM): GOSS is a method exclusively employed in LightGBM [6]. The idea behind the Goss method is that all data points contribution to training is not equal as explained before if the data point has a small gradient it tends to be more trained, i.e., close to local minima. Thus, the algorithm will be more efficient if it focuses on data points that have larger gradients. However, when we use this observation straightforward by ignoring any data point with small gradients while computing the best split this will increase the risk of bias in sampling and change the data distribution. For example, if we have a dataset with all the data points at a young age are tended to be less well trained, the sampled data taken from this dataset will bias toward a younger age. Thus, the algorithm will choose a split that is younger – tend to a young age – instead of the optimal value.

To overcome this problem, LightGBM employs an additional technique that uses random sampling to select from the data points with small gradients alongside all data points with large gradients. This technique helps the algorithm to maintain a sample with a biased towards data with large gradients alongside data points with small gradients. Then while computing the contribution of sample to the change in the loss, LightGBM will give more weights to the samples with small gradients.

Feature Combination (CatBoost): CatBoost uses a novel way to find the best split for the current tree [13]. Namely, the combinations are been considered in a greedy fashion. For the first split in the tree, the algorithm doesn't consider combinations. For the next splits, all categorical features and combinations that are present in the current tree are been combined with all of the categorical features in the dataset. CatBoost converts all combination values into numbers very quickly. Furthermore, CatBoost is able to generate numerical and categorical feature combinations in the following way: all the selected splits in the tree are treated as a categorical feature with two values that are been used in combinations similar to the categorical ones.

1.3. Handling sparse inputs

Since those algorithms often are used on text data or vectorized tabular. Thus, the inputs usually are tended to be sparse. Because most of the values of a spare feature will be 0, it will be wasteful to go through all of its values. XGBoost way of dealing with this problem is by ignoring all 0 features when computing the split, then takes all the data with missing values and allocate them to either side of the split. And this helps to reduce the loss more as shown in Figure 5. This process will speed the training up by reducing the number of samples required to be used in evaluating each split [6,11].

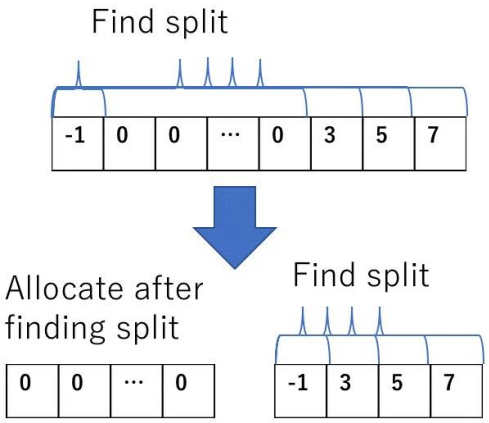

Figure 5. Dealing with sparse data.

Both XGBoost and LightGBM are treating the missing values by ignoring them while computing the best split, then allocates them to either side that reduces the loss the most which is by coincidence, the same way that XGBoost is handing spares data. LightGBM has an option called zero_as_missing if it is enabled (set to True) will consider all zero values as missing. But is not enabled by default [14]. Unlike XGBoost and LightGBM, CatBoost has no support for sparse data until now. The algorithm team is working on bringing this feature to the algorithm though [15]. Therefore, CatBoost requires handling the sparse data before feeding it to the model. In case of missing values processing mode in CatBoost depends on the feature type:

- Numerical features: The default processing mode is Min. There are other ways provided like; Max and forbidden. in order to CatBoost
- Categorical features: CatBoost does not process categorical features in any specific way.

1.4. Exclusive Feature Bundling (LightGBM)

LightGBM has a novel method to effectively reduce the number of features that utilize sparsity in large datasets. In large datasets, the sparsity of features means that some features are never non-zero together this observation is the essential building block for EFG in lighGBM [6]. For example, the word "Java" and "vein" might never be in the same document. Therefore, these two features can be bundled together into a single feature without any loss in the information of the data. Assume that the tf-idf (term frequency-inverse document frequency) score for "Java" is between 0 - 15 and the tf-idf score for "vein" is between 0 - 30. In this case, the feature:

```
Java   if vein = 0  else  vein += 15
```

Would be between 0 - 45, also this score can easily be converted back to its original scores. Unfortunately, finding the most efficient bundle is an NP-hard problem. Therefore, the creators of this algorithm use an approximate algorithm that has toleration with a certain degree for overlapping γ between the non-zero elements within a feature bundle.

1.5. Treating the Categorical Variables

LightGBM has an implementation called categorical_feature for handling categorical features. The input of this parameter is the names of the categorical features. While it doesn't use the one-hot coding method it is way faster than one-hot coding. But all the categorical features must be converted to integers before passing it through categorical_feature parameter. And for finding the split value of the categorical features LightGBM uses a distinctive algorithm [16].
Example for specifying feature names and categorical features in LightGBM:

```
Train_df = lgbm.Dataset(df ,label=target, feature_name=['ft1', 'ft2', 'ft3'],
                            categorical_feature=['ft3'])
```

CatBoost has a novel method to handle categorical features implemented on the algorithm. It uses the categorical features to construct new numerical features that are based on categorical features and their combinations. As for categorical features, CatBoost uses one-hot encoding for them with different values depending on the training mode. Categorical data can be specified in cat_features argument [17]. One thing to keep in mind that in order for CatBoost to consider the categorical features as categorical we have to pass all of them in the cat_features argument if we don't do that CatBoost will consider all the features in the dataset as numerical variables. Moreover, even columns with integer-type values also will be considered as numerical values.

XGBoost unlike LightGBM and CatBoost, it doesn't have any implementations for handling categorical features. Similar to Random Forest, XGBoost accepts only numerical features as an input. Thus, all the categorical features have to be encoded with methods like label encoding or one-hot encoding before feeding them to the algorithm.

1.6. Categorical Boosting (CatBoost)

CatBoost is built on the idea of categorical boosting. When the algorithm is building the tree before each split is selected, it transforms all the categorical features into numerical. This transformation is done by applying several statistics on both combinations of only categorical features and combinations of categorical and numerical features together. The transformation method used to transform the categorical features into numerical in general includes the following:

1. Randomly permutating the input objects set.
2. The label value is been converted from a floating-point into an integer. This process differs based on the problem that is being solved (this is determined by the loss function that is being used).
3. Transforming categorical features into numerical features using the following:

$$\text{avg\_target} = \frac{\text{CountInClass} + \text{Prior}}{\text{TotalCount} + 1} \tag{1}$$

CountInClass refers to how many the label values for the objects of the value of the current categorical feature was equal to 1. **Prior** is an initial value for the numerator that has been determined based on the starting parameters. **TotalCount** refers to the total number of objects (up to the current one) with a categorical feature value matching the current one.

Therefore, the algorithm is assigning a numerical feature to each of the values of the categorical features or the value of the feature combination [17].

1.7 Hyperparameters Similarities

Due to the similar foundation of all the three algorithms, they have similar hyperparameters. The similar hypermeters are showing in Table 1 [4]:

Table 1. Table shows the hyperparameters similarities between the three algorithms.

Function	XGBoost	CatBoost	LightGBM
Important arameters which control overtting	**learning_rate or eta:** Optimal values lie between 0.01-0.2. **max_depth** **min_child_weight:** similar to min child leaf; default is 1.	**learning_rate** **depth:** Value can be any integer up to 16. Recommended [1 to 10]. No such feature like min child weight. **L2-leaf-reg:** L2 regularization coefficient is used for the calculation of leaf value (positive integer)	**learning_rate** **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 tree) which should be smaller than 2^(max depth) it is very important parameter. **min_data_in_leaf:** default = 20.
Parameters for categorical values	Not Available	**Cat_features:** It represents the index of the categorical features. **One_hot_max_size:** Use to specifiy the maximum number of different values while performing one-hot encoding for all features (max - 255)	**categorical-feature:** Use to specify which of the features is categorical during training the model.
Parameters for controlling speed	**colsample_bytree:** subsample ratio of columns. **subsample:** subsample ratio of the training instance. **n_estimators:** maximum number of decision trees; high value can lead to over fitting	**rsm:** Random subspace method. The percentage of features to use at each split selection. No such parameter to subset data. **iterations:** specify the upper limit of the trees that can be built; high value can lead to overfitting.	**feature_fraction:** fraction of features to be taken for each iteration. **bagging_fraction:** specify the data fraction which will be used in each iteration and it is used to avoid overfitting and speed up the training. **Num_iterations:** specify how many boosting iterations to be performed; default = 100.

2. Materials and Method

2.1. Datasets

To be able to evaluate the performance of the three algorithms accurately in both classification and regression five datasets have been used in this paper. The datasets have a different size in features and entries to ensure the diversity of the datasets.

The US Adult Census Dataset (Classification)

The US Adult Census dataset has been extracted from the 1994 US Census database. This dataset includes more than 48 thousand entries about citizens in various countries with 15 features (columns). The objective here is to predict the income of the citizen-based on the information in the other columns (features) [18].

WHO Life Expectancy Dataset (Regression)

A dataset repository that operates under the World Health Organization (WHO) called Global Health Observatory (GHO) monitors the health status along with other related factors around the world. The life expectancy dataset is a combination of the GHO datasets and the United Nation datasets. The life expectancy includes data about health factors for 193 countries that have been collected from Global Health Observatory (GHO). Next, the corresponding economic data have been collected for UN datasets. The resulting dataset has been processed and the missing data handled with the help of the Missmap command in R software. The final dataset consists of 22 Columns (features) and 2938 rows (entries) which meant 20 predicting variables [19].

Data Hackathon 3.x (Classifications)
Data Hackathon 3.x dataset consists of 87020 entries of client's data of banks in India with 26 features and the goal is to predict if the client will disburse or not.

Bank Customer Churn (Classifications)
This dataset is also the client's data of bank in Europe which consists of 10000 entries and 14 features with a goal of predicting whether the client will leave the bank or not.

Bank Customer Churn (Classifications)
The Framingham Hearts study which began in 1948 and has uncovered numerous associations or risk factors relating to coronary artery disease. This portion of the data set includes 4,238 subjects, each with 15 descriptive measures (features) such as cholesterol, blood pressure, and heart rate in addition to an output label - if a diagnosis of CHD (Coronary Heart Disease) was made over 10 years [20].

2.2. Method
The evaluation process will consist of four main categories speed, accuracy, reliability, and ease of use. The three algorithms will be tested with the default and tuned hyperparameters to show which of them better in both cases. The tests have been done with the latest python packages of the algorithm on the same computer in the same conditions. The tuning process has been done many times to find the best hyperparameters for each algorithm separately. In case of the speed test each case of the algorithms has been done 10 times with the same hyperparameters then the average of them has been taken. The results will be shown in a table for better comprehension of the differences. For the third category ease of use, we evaluate it by considering the number of hyperparameters for tuning the algorithm and how much the algorithm performs with the default hyperparameters. Finally, to measure the reliability we consider how consistent the algorithm performs across all datasets that we use.

2.3. Computer specifications
All the test and tuning process has been done in a laptop with a mid-range specification to demonstrate how well the algorithms perform and to make the differences in speed more noticeable. We intended to use mid-range computer because most of the researchers and students use computers with similar specifications to do most of their programming work. Thus, we wanted to see how they going to perform on this kind of hardware.
The computer which has been used to perform all the tests has a Core i5-6200u processor paired with 6 GB of DDR3 Ram and NVIDIA GeForce 940M graphics card with 2 GB of memory and 256 GB of SATA 3 SSD for storage. As for software Table 2 shows the versions of the programs and libraries.

Table 2. Software versions.

Python	3.7
XGBoost	0.80
CatBoost	0.14.2
LightGBM	2.2.2

3. Results

3.1. The US Adult Census dataset (classification) results

Table 3. The US Adult Census Dataset's accuracy and speed results of the four algorithms.

	XGBoost		**XGBoost - hist**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	01.104954 Sec	13.84216 Sec	00.367410 Sec	01.940795 Sec
Accuracy	86.794423	87.599042	86.788281	87.777164
	LightGBM		**CatBoost**	
	Default	Tuned	Default	Tuned
Speed	00.361316 Sec	00.653143 Sec	94.158556 Sec	27.683200 Sec
Accuracy	87.341073	87.660463	87.482341	87.580615

Table 3 shows two metrics of evaluation speed and accuracy both with the default and tuned hyperparameters for each algorithm. From the results, we can infer that all algorithms perform well with very good accuracy with a small margin between the accuracies. however, the XGBoost-hist algorithm with tuned hyperparameters has the best accuracy. As for the speed, there is a big difference in the result ranging from the fastest LightGBM with default hyperparameter with only 00.361316 sec to complete the training process to the slowest CatBoost with default hyperparameter with a whopping 94.158556 sec to complete the same task. Furthermore, we notice that

XGBoost and LightGBM speed decreased after tuning them whereas, XGBoost - hist and CatBoost speed increased with tuning especially with CatBoost we were able to speed it up to 4 times with better accuracy.

3.2. WHO Life Expectancy dataset (regression) results

Table 4. WHO Life Expectancy Dataset's accuracy and speed results of the four algorithms.

	XGBoost		**XGBoost - hist**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.943869 Sec	08.87374 Sec	00.351246 Sec	06.529343 Sec
Accuracy	94.248202	96.579688	94.199483	96.559307
	LightGBM		**CatBoost**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.311644 Sec	01.374218 Sec	36.196142 Sec	17.62043 Sec
Accuracy	94.359751	95.877220	92.017064	96.132491

In Table 4 we have the result for the WHO Life Expectancy Dataset. We notice the same pattern for the accuracies but with a bigger margin with the worst accuracy belongs to CatBoost with the default hyperparameter and the best belongs to XGBoost with tuned hyperparameter. Regarding speed, the same can be said. LightGBM with default hyperparameter is the fastest and CatBoost with default hyperparameter as expected is the slowest. Moreover, the tuning of XGBoost, XGBoost – hist, and LightGBM slow down their speed in contrary for CatBoost the tuning speeds it up.

3.3. Data Hackathon 3.x dataset (classifications) results.

Table 5. Data Hackathon 3.x dataset's accuracy and speed results of the four algorithms.

	XGBoost		**XGBoost - hist**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	01.782270 Sec	00.864689 Sec	00.465752 Sec	00.403920 Sec
Accuracy	98.501494	98.501494	98.501494	98.501494
	LightGBM		**CatBoost**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.813822 Sec	00.395931 Sec	68.196164 Sec	33.320850 Sec
Accuracy	98.483107	98.501494	98.501494	98.501494

Similarly, Table 5 has the Data Hackathon 3.x datasets' results. We infer that all algorithms have excellent accuracy with a very small margin. All of the algorithms have the same accuracy except LightGBM with default hyperparameter has slightly lower accuracy. As expected, LightGBM with tuned hyperparameter is the fastest whereas CatBoost with default hyperparameter is the slowest. However, the speed of all algorithms increased after tuning almost double except XGBoost - hist with only a 15% increase.

3.4. Bank Customer Churn dataset (classifications) results

Table 6. Bank Customer Churn Dataset's accuracy and speed results of the four algorithms.

	XGBoost		**XGBoost - hist**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.75184 Sec	00.71689 Sec	00.300096 Sec	00.727746 Sec
Accuracy	87.15	87.15	86.90	86.95
	LightGBM		**CatBoost**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.194973 Sec	00.113583 Sec	33.449706 Sec	09.558820 Sec
Accuracy	86.00	86.85	86.65	87.20

Table 6 illustrates the Bank Customer Churn datasets' results. We infer that all algorithms have very good accuracy with a small margin with CatBoost with tuned hyperparameter has the best accuracy followed by XGBoost with a small difference and LightGBM has the worst accuracy. Following the same pattern, CatBoost with default hyperparameter is the slowest, and LightGBM with Tuned hyperparameter is the fastest. Moreover, tuning algorithms speed them up except XGBoost - hist which made it more than 2 times slower. As expected, CatBoosts' speed significantly increased with tuning almost 3.5 times faster.

3.5. Framingham Heart Study dataset (classifications) results

Table 7. Framingham Heart Study Dataset's accuracy and speed results of the four algorithms.

	XGBoost		XGBoost - hist	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.173532 Sec	00.122078 Sec	00.150508 Sec	00.98946 Sec
Accuracy	85.519126	85.792350	85.382514	85.519126
	LightGBM		**CatBoost**	
	Default	**Tuned**	**Default**	**Tuned**
Speed	00.146186 Sec	00.150189 Sec	52.861558 Sec	10.478870 Sec
Accuracy	84.562842	85.519126	85.245902	85.792350

Similarly, Table 7 shows Framingham Heart Study Datasets' results. By glancing at the results, we can notice the pattern of the three algorithms' performance continues. All of them have very good accuracy with a small difference. XGBoost and CatBoost with tuned hyperparameters have the best accuracy whereas LightGBM with default hyperparameters has the worst. As for the speed, the fastest is XGBoost - hist with tuned hyperparameters followed by LightGBM with tuned hyperparameters. As expected, CatBoost with default hyperparameters is the slowest, and we were able to speed it up significantly by more than 5 times.
After analyzing the results above, we found that the fastest algorithm is LightGBM, followed by XGBoost with histogram implementation with a small difference, and the slowest was CatBoost, which takes significantly more time than the others as shown in Figure 6. Furthermore, we have noticed that CatBoost can be speeded up by tuning the hyperparameters up to 5 times in our tests with better accuracy. However, it's still significantly slower compared to its analogs, and this pattern continues throughout all the tests. The team that works on CatBoost has been working to make the algorithm faster, so the next versions of the algorithm may be faster. Moreover, the speed of LightGBM is better when we preprocess the categorical data by transforming them into a binary (Boolean) matrix.

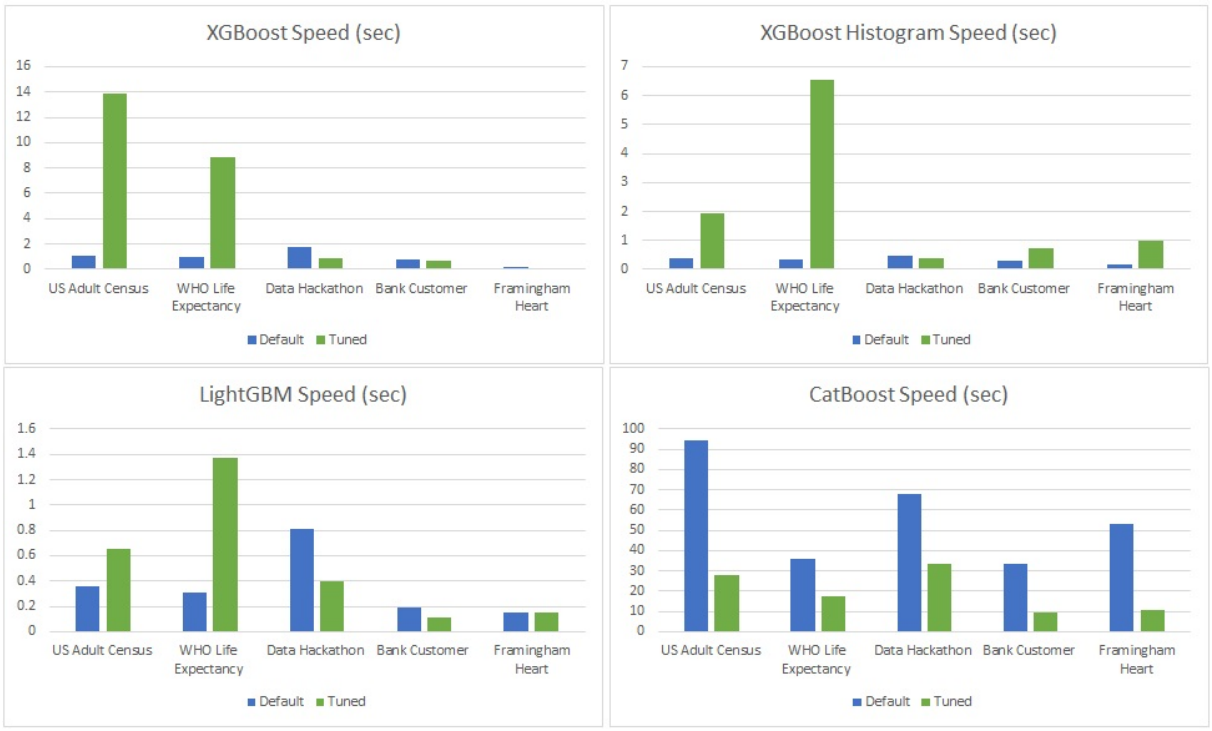

Figure 6. Graphs show the speed of each algorithm with each dataset.

In case of accuracy, all algorithms perform well with small differences, both with the default and tuned hyperparameters. However, we notice that LightGBM has slightly better accuracy with default hyperparameters followed CatBoost and XGBoost as shown in Figure 7.

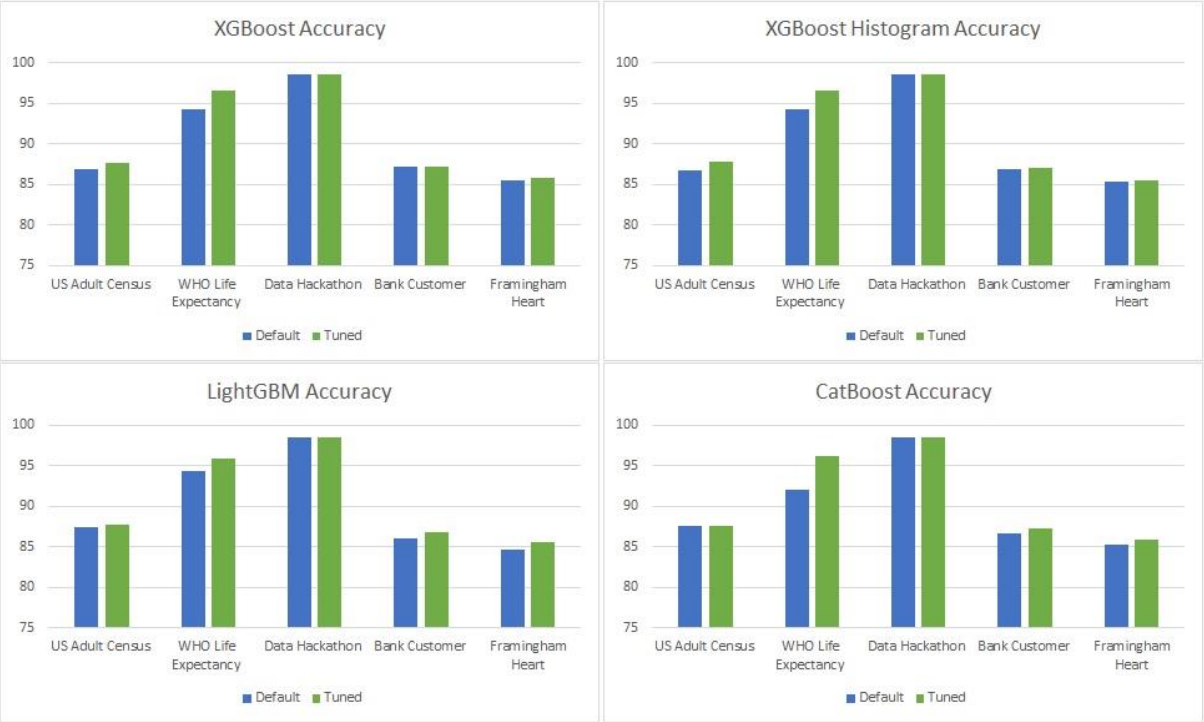

Figure 7. Graphs show the accuracy of each algorithm with each dataset.

As for reliability, LightGBM and XGBoost have consistent results in both default and tuned hyperparameters with small differences. On the contrary, CatBoost had a noticeable difference in its results.

Last the ease of use, the three algorithms have an official package that makes them easy to use, but due to the way the CatBoost works makes it the easiest algorithm to use because we don't have to preprocess the categorical data. Moreover, CatBoost has fewer hyperparameters to tune. The second easiest is LightGBM because it can handle categorical data as long as they are in numerical form. The last algorithm is XGBoost, which requires transforming the categorical data to a binary (Boolean) matrix. We also found that CatBoost in most cases doesn't use all of the CPU resources in the training process, unlike the XGBoost and LightGBM.

4. Discussion and Conlusion

Our results show that LightGBM is the most balanced performance algorithm. It produced a very good accuracy across the five datasets with excellent speed, in three datasets was the fastest and in the other was the second-fastest. Moreover, it has a consistent result with all of the datasets. Moreover, since LightGBM can handle categorical data makes it easy to use. As for XGBoost, while the default method - exact - is slower than LightGBM, the histogram-based method has an equivalent speed to LightGBM with almost the same accuracy. However, since it requires preprocessing the categorical data makes it slightly harder than LightGBM. Finally, with the ability to handle categorical data without any preprocessing and fewer hyperparameters to tune CatBoost is the easiest to use. However, CatBoost accuracy and speed are less than its analogs and vary a lot. Although we were able to speed it up to 5 times, it still much slower than the other two. Finally, those results show that XGBoost is not the only GBDT algorithm with state-of-the-art performance. LightGBM and CatBoost as well are capable algorithms with novel ideas.

Conflicts of Interest

The authors of this study declare that there is no conflict of interest.

References

[1] J. H. Friedman, "Stochastic gradient boosting", Computational statistics & data analysis, vol 38, no. 4, 367-378, (2002).

[2] J. H. Friedman, "Greedy function approximation: a gradient boosting machine". Annals of statistics, 1189-1232, (2001).

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., "Scikit-learn: Machine learning in python," Journal of machine learning research, vol. 12, no. Oct, 2825-2830, (2011).

[4] A. Swalin. "Catboost vs. Lightgbm vs. Xgboost". (2018), [Online]. Available: https://towardsdatascience.com/catboost-vs-light-gbm-vs-xgboost-5f93620723db. Access date: 19.03.2019.

[5] X. Team. "Introduction to boosted trees". (2019), [Online]. Available: https://xgboost.readthedocs.io/en/latest/tutorials/model.html. Access date: 19.03.2019.

[6] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, Lightgbm: A highly efficient gradient boosting decision tree," in Advances in Neural Information Processing Systems, , 3146-3154, (2017).

[7] P. Langley and S. Sage, "Oblivious decision trees and abstract cases" Working notes of the AAAI-94 workshop on case-based reasoning. Seattle, WA, 113-117, (1994).

[8] R. Kohavi and C.-H. Li, "Oblivious decision trees, graphs, and top-down pruning," IJCAI. Citeseer, 1071-1079, (1995).

[9] V. Ershov. "Catboost enables fast gradient boosting on decision trees using gpus". (2018), [Online]. Available: https://devblogs.nvidia.com/catboost-fast-gradientboosting-decision-trees/. Access date: 13.04.2019.

[10] C. Team. "How training is performed". (2018), [Online]. Available: https://catboost.ai/docs/concepts/algorithm-main-stages.html. Access date: 23.05.2019.

[11] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system", Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. ACM, 785-794, (2016).

[12] X. Team. "Xgboost parameters". (2019) [Online]. Available: https://xgboost.readthedocs.io/en/latest/parameter.html. Access date: 19.03.2019.

[13] A. V. Dorogush, V. Ershov, and A. Gulin, "Catboost: gradient boosting with categorical features support", arXiv preprint arXiv:1810.11363, 2018.

[14] L. Team. "Advanced topics, missing value handle". (2019), [Online]. Available: https://lightgbm.readthedocs.io/en/latest/Advanced-Topics.html#missing-value-handle . Access date: 11.04.2019.

[15] A. V. Dorogush, "Catboost - the new generation of gradient boosting," PyData, (2018), [Online]. Available: https://www.youtube.com/watch?v=8o0e-r0B5xQ .Access date: 23.05.2019.

[16] L. Team. "Advanced topics, categorical feature support" (2019) [Online]. Available: https://lightgbm.readthedocs.io/en/latest/Advanced-Topics.html#categorical-feature-support. Access date: 11.04.2019.

[17] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "Catboost: unbiased boosting with categorical features", Advances in Neural Information Processing Systems, 2018, 6638-6648, (2018).

[18] C. database. Adult data set. (1996) [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Adult. Access date: 18.03.2019.

[19] WHO. "Life expectancy" (2018) [Online]. Available: https://www.kaggle.com/kumarajarshi/life-expectancy-who. Access date: 18.03.2019.

[20] L. The National Heart and B. I. (NHLBI). "Framingham heart study", (2018). [Online]. Available: https://www.nhlbi.nih.gov/science/framingham-heartstudy-fhs. Access date: 18.03.2019.