If things don’t go your way in [predictive modeling](https://www.analyticsvidhya.com/blog/2021/06/what-is-predictive-analytics-an-introductory-guide-for-data-science-beginners/), use XGboost. [XGBoost algorithm](https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/" \t "_blank)has become the ultimate weapon of many data scientists. It’s a highly sophisticated algorithm, powerful enough to deal with all sorts of irregularities of data. It uses parallel computation in which multiple decision trees are trained in parallel to find the final prediction. This article is best suited to people who are new to XGBoost. We’ll learn the art of XGBoost parameters tuning and XGBoost hyperparameter tuning. Also, we’ll practice this algorithm using a training data set in Python.

# What is XGBoost?

Building a machine learning model using XGBoost is easy. But, improving the model using XGBoost is difficult (at least I struggled a lot). This algorithm uses multiple parameters. To improve the model, parameter tuning is a must to get the best parameter values. It is very difficult to get answers to practical questions like – Which set of parameters should you fine-tune? What is the ideal value of these parameters to obtain optimal output?

**XGBoost (eXtreme Gradient Boosting)** is an advanced implementation of a gradient boosting algorithm. Since I covered Gradient Boosting Machine in detail in my previous article – [Complete Guide to Parameter Tuning in Gradient Boosting (GBM) in Python](https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/), I highly recommend going through that before reading further. It will help you bolster your understanding of boosting in general and parameter tuning for GBM.

Sample Project to Apply XGBoostProblem StatementHR analytics is revolutionizing the way human resources departments operate, leading to higher efficiency and better results overall. Human resources have been using analytics for years.However, the collection, processing, and analysis of data have been largely manual, and given the nature of human resources dynamics and HR KPIs, the approach has been constraining HR. Therefore, it is surprising that HR departments woke up to the utility of [machine learning](https://www.analyticsvidhya.com/machine-learning/?utm_source=blog&utm_medium=parameter-tuning-xgboost) so late in the game. Here is an opportunity to try predictive analytics in identifying the employees most likely to get promoted.[Practice Now](http://datahack.analyticsvidhya.com/contest/wns-analytics-hackathon-2018-1/?utm_source=av_blog&utm_medium=practice_blog_xgboost)

# Advantages of XGBoost

I’ve always admired the boosting capabilities that this algorithm infuses into a predictive model. When I explored more about its performance and the science behind its high accuracy, I discovered many advantages:

**Regularization**

Standard GBM implementation has no [regularization](https://www.analyticsvidhya.com/blog/2015/02/avoid-over-fitting-regularization/) like XGBoost; therefore, it also helps to reduce overfitting. In fact, XGBoost is also known as a ‘**regularized boosting**‘ technique.



**Ready to start your data science journey?**

**Master 23+ tools & learn with guided projects with the Certified AI & ML BlackBelt Plus Program**

[Enroll Now](https://blackbelt.analyticsvidhya.com/plus?utm_source=blog_india&utm_medium=desktop_banner_between_articles&utm_campaign=01-Oct-2023%7C%7C&utm_content=tools)

**Parallel Processing**

XGBoost implements parallel processing and is **blazingly faster** as compared to GBM.

But hang on, we know that [boosting](https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/) is a sequential process so how can it be parallelized? We know that each tree can be built only after the previous one, so what stops us from making a tree using all cores? I hope you get where I’m coming from. Check [this link](http://zhanpengfang.github.io/418home.html) out to explore further. XGBoost also supports implementation on Hadoop.

**High Flexibility**

XGBoost allows users to define **custom optimization objectives and evaluation criteria**. This adds a whole new dimension to the model and there is no limit to what we can do.

**Handling Missing Values**

XGBoost has an in-built routine to handle missing values. The user is required to supply a different value than other observations and pass that as a parameter. XGBoost tries different things as it encounters a missing value on each node and learns which path to take for missing values in the future.

**Tree Pruning**

A GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a **greedy algorithm**. XGBoost, on the other hand, makes **splits up to the max\_depth** specified and then starts **pruning** the tree backward and removing splits beyond which there is no positive gain.

Another advantage is that sometimes a split of negative loss, say -2, may be followed by a split of positive loss +10. GBM would stop as it encounters -2. But XGBoost will go deeper, and it will see a combined effect of +8 of the split and keep both.

**Built-in Cross-Validation**

XGBoost allows the user to run a **cross-validation at each iteration** of the boosting process and thus, it is easy to get the exact optimum number of boosting iterations in a single run. This is unlike GBM, where we have to run a grid search, and only limited values can be tested.

**Continue on the Existing Model**

Users can start training an XGBoost model from its last iteration of the previous run. This can be of significant advantage in certain specific applications. GBM implementation of sklearn also has this feature, so they are even on this point.

I hope now you understand the sheer power XGBoost algorithm. Note that these are the points that I could muster. Do you know a few more? Feel free to drop a comment below, and I will update the list.

# What are XGBoost Parameters?

The overall parameters have been divided into 3 categories by XGBoost authors:

1. **General Parameters:** Guide the overall functioning
2. **Booster Parameters:** Guide the individual booster (tree/regression) at each step
3. **Learning Task Parameters:** Guide the optimization performed

**Must Read:**[**Complete Machine Learning Guide to Parameter Tuning in Gradient Boosting (GBM) in Python**](https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/)

General Parameters

These define the overall functionality of XGBoost.

1. **booster [default=gbtree]**
   * Select the type of model to run at each iteration. It has 2 options:
     + gbtree: tree-based models
     + gblinear: linear models
2. **silent [default=0]**
   * Silent mode is activated is set to 1, i.e., no running messages will be printed.
   * It’s generally good to keep it 0 as the messages might help in understanding the model.
3. **nthread [default to the maximum number of threads available if not set]**
   * This is used for parallel processing, and the number of cores in the system should be entered
   * If you wish to run on all cores, the value should not be entered, and the algorithm will detect it automatically

There are 2 more parameters that are set automatically by XGBoost, and you need not worry about them. Let’s move on to Booster parameters.

Booster Parameters

Though there are 2 types of boosters, I’ll consider only **tree booster** here because it always outperforms the linear booster, and thus the latter is rarely used.

1. **eta [default=0.3]**
   * Analogous to the learning rate in GBM
   * Makes the model more robust by shrinking the weights on each step
   * Typical final values to be used: 0.01-0.2
2. **min\_child\_weight [default=1]**
   * 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 the min “sum of weights” of observations, while GBM has the min “number of observations”.
   * Used to control over-fitting. Higher values prevent a model from learning relations that 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.
3. **max\_depth [default=6]**
   * The maximum depth of a tree is the same as GBM.
   * Used to control over-fitting as higher depth will allow the model to learn relations very specific to a particular sample.
   * It should be tuned using CV.
   * Typical values: 3-10
4. **max\_leaf\_nodes**
   * The maximum number of terminal nodes or leaves in a tree.
   * It can be defined in place of max\_depth. Since binary trees are created, a depth of ‘n’ would produce a maximum of 2^n leaves.
   * If this is defined, GBM will ignore max\_depth.
5. **gamma [default=0]**
   * 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.
   * Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
6. **max\_delta\_step [default=0]**
   * In the maximum delta step, we allow each tree’s weight estimation to be. If the value is set to 0, there is no constraint. If it is set to a positive value, it can help make the update step more conservative.
   * Usually, this parameter is not needed, but it might help in logistic regression when the class is extremely imbalanced.
   * This is generally not used, but you can explore further if you wish.
7. **subsample [default=1]**
   * Same as the subsample of GBM. Denotes the fraction of observations to be random samples for each tree.
   * Lower values make the algorithm more conservative and prevent overfitting, but too small values might lead to under-fitting.
   * Typical values: 0.5-1
8. **colsample\_bytree [default=1]**
   * Similar to max\_features in GBM. Denotes the fraction of columns to be random samples for each tree.
   * Typical values: 0.5-1
9. **colsample\_bylevel [default=1]**
   * Denotes the subsample ratio of columns for each split in each level.
   * I don’t use this often because subsample and colsample\_bytree will do your job. but you can explore further if you feel so.
10. **lambda [default=1]**
    * L2 regularization term on weights (analogous to Ridge regression)
    * This is used to handle the regularization part of XGBoost. Though many data scientists don’t use it often, it should be explored to reduce overfitting.
11. **alpha [default=0]**
    * L1 regularization term on weight (analogous to Lasso regression)
    * It can be used in case of very high dimensionality so that the algorithm runs faster when implemented
12. **scale\_pos\_weight [default=1]**
    * A value greater than 0 should be used in case of high-class imbalance as it helps in faster convergence.

Learning Task Parameters

These parameters are used to define the optimization objective and the metric to be calculated at each step.

1. **objective [default=reg:linear]**
   * This defines the loss function to be minimized. Mostly used values are:
     + **binary: logistic** –logistic regression for binary classification returns predicted probability (not class)
     + **multi: softmax** –multiclass classification using the softmax objective, returns predicted class (not probabilities)
       - you also need to set an additional **num\_class**(number of classes) parameter defining the number of unique classes
     + **multi: softprob** –same as softmax, but returns predicted probability of each data point belonging to each class.
2. **eval\_metric [ default according to objective ]**
   * The evaluation metrics are to be used for validation data.
   * The default values are rmse for regression and error for classification.
   * Typical values are:
     + **rmse** – root mean square error
     + **mae** – mean absolute error
     + **logloss** – negative log-likelihood
     + **error** – Binary classification error rate (0.5 thresholds)
     + **merror** – Multiclass classification error rate
     + **mlogloss** – Multiclass logloss
     + **auc:** Area under the curve
3. **seed [default=0]**
   * The random number seed.
   * It can be used for generating reproducible results and also for parameter tuning.

If you’ve been using Scikit-Learn till now, these parameter names might not look familiar. The good news is that the xgboost module in python has an sklearn wrapper called XGBClassifier parameters. It uses the sklearn style naming convention. The parameters names that will change are:

1. eta –> learning\_rate
2. lambda –> reg\_lambda
3. alpha –> reg\_alpha

You must be wondering why we have defined everything except something similar to the “n\_estimators” parameter in GBM. Well, this exists as a parameter in XGBClassifier. However, it has to be passed as “num\_boosting\_rounds” while calling the fit function in the standard xgboost implementation.

# Hyper parameters tuning with optuna

Xgb.train takes dmatrix data which is fast and also for xgb.train use num\_boost\_rounds which is same as estimators

def cross\_validation(X,y,best\_params,n\_splits, xtest=None):

    skf = StratifiedKFold(n\_splits=n\_splits, random\_state=42, shuffle=True)

    acc\_scr = []

    f1\_scr = []

    auc\_scr = []

    eval\_results\_ = {}

    if xtest is not None:

        preds = np.zeros(len(xtest))

    else:

        preds = None

    for i, (train\_index, test\_index) in enumerate(skf.split(X, y)):

        X\_train, X\_test = X.iloc[train\_index], X.iloc[test\_index]

        y\_train, y\_test = y.iloc[train\_index], y.iloc[test\_index]

*# Dmatrix train and Test dataset*

        dtrain = xgb.DMatrix(X\_train, y\_train)

        dtest  = xgb.DMatrix(X\_test, y\_test)

*# specify validations set to watch performance*

        watchlist = [(dtest, "eval"), (dtrain, "train")]

*# number of boosting rounds*

        num\_round = best\_params["n\_estimators"]

        eval\_results\_[i] = {}

*# Specify which dataset and which metric should be used for early stopping.*

        early\_stop = xgb.callback.EarlyStopping(rounds=50,

                                                data\_name='eval',

                                                save\_best=True)

        model = xgb.train(best\_params,

                          dtrain,

                          num\_boost\_round = num\_round,

                          verbose\_eval  = False,

                          evals\_result  = eval\_results\_[i],

                          evals = watchlist,

                          callbacks     = [early\_stop])

        y\_preds = np.rint(model.predict(dtest))

        dxtest = xgb.DMatrix(xtest)

        if xtest is not None:

            test\_preds = model.predict(dxtest) / n\_splits

            if preds is None:

                preds = test\_preds

            else:

                preds += test\_preds

        acc\_scr.append(accuracy\_score(y\_test, y\_preds))

        f1\_scr.append(f1\_score(y\_test, y\_preds))

        auc\_scr.append(roc\_auc\_score(y\_test, model.predict(dtest)))

    avg\_acc = round(np.mean(acc\_scr), 4)

    avg\_f1 = round(np.mean(f1\_scr), 4)

    avg\_roc = round(np.mean(auc\_scr), 4)

    print(f"Average of {n\_splits} splits accuracy score : {avg\_acc:.4f} | f1 score : {avg\_f1:.4f} | roc score : {avg\_roc:.4f}")

    return avg\_roc, preds

%%time

def objective\_pr(trial):

    model\_params = {

*# Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug)*

        'verbosity': 0,

*# n\_estimators is num\_boosting\_rounds*

        'n\_estimators': trial.suggest\_int('n\_estimators', 50, 1500, step = 10),

        'learning\_rate': trial.suggest\_float('learning\_rate', 1e-7, 1e-1),

*#Makes the model more robust by shrinking the weights on each step*

        'eta': trial.suggest\_float('eta', 0.01, 0.3),

        'max\_depth': trial.suggest\_int('max\_depth', 3, 20),

*#Maximum number of nodes to be added*

        'max\_leaves':trial.suggest\_int('max\_leaves', 0, 20),

*# bins for continous features*

        'max\_bin': trial.suggest\_int('max\_bin', 150, 500, step = 10),

*#minimum sum of weights of all observations required in a child*

        'min\_child\_weight': trial.suggest\_int('min\_child\_weight', 1, 5),

*#0 to 1, 0.5 means 50% random sample for grow trees*

        'subsample': trial.suggest\_float('subsample', 0.1, 1.0, step = 0.1),

*#0 to 1, 0.5 means 50% columns for trees like max\_features*

        'colsample\_bytree': trial.suggest\_float('colsample\_bytree', 0.1, 1.0,step = 0.1),

*# L1 regularization*

        'alpha': trial.suggest\_float('alpha', 1e-5, 1e2),

*# L2 regularization*

        'lambda': trial.suggest\_float('lambda', 1e-5, 1e2),

*# Loss function*

        'objective': 'binary:logistic',

*# evaluation metric on test data*

        'eval\_metric': 'auc',

        'booster':trial.suggest\_categorical("booster", ["dart", "gbtree",'gblinear']),

        'tree\_method':trial.suggest\_categorical("tree\_method", ["approx", "hist"]),

        'grow\_policy':trial.suggest\_categorical("grow\_policy", ["depthwise", "lossguide"]),

        'random\_state' : 42

    }

    scores = cross\_validation(X, y, model\_params, 5, xtest=None)[0]

    return np.mean(scores)

study = optuna.create\_study(direction='maximize')

study.optimize(objective\_pr, n\_trials=20)

best\_params = study.best\_params