# LightGBM Hyperparameters

2.1. n\_estimators  
The number of individual trees in the model. Each tree uses random features to improve the generalization of the model since each tree is fitted on a random subset of the features.

Tip: This can sometimes be less than the number of epochs if early\_stopping is used during training.  
i.e. if n\_estimators = 10000 but early\_stopping is used, if the stopping condition stops the training at epoch 7300. n\_estimators will be 7300.

**2.2. learning\_rate**  
The shrinkage of the prediction of the boosting model at each split in the next tree. learning\_rate can be viewed as the step size used in the gradient descent method. If your learning rate is too large you might skip the optimum or move into unstable directions. If your learning\_rate is too small you will take longer to find the optimum.

Tip: A model might generalize well to the test set because of the regularization (L1 and L2 loss cause the model to fit to the training set a bit looser, meaning the residual error isn't perfectly 0) but then if that same model is used to predict new data, the model might be too general/wide, over fitting to the test set and underfitting to the new data.

**2.3. boosting\_type**  
This specifies the shape of the decision tree to be used as a base model in the ensemble. You can find more details here. List of Parameter Values LightGBM uses the leaf-wise algorithm while other packages use level-wise algorithms. Nonetheless, LightGBM might sometimes use a level-wise algorithm that will perform better with smaller datasets.

Tip: Can be used to change the dense matrix to sparse matric. Levels are more regularized than leaves.  
3 boosting types are supported by LightGBM:

* gbdt: Traditional Gradient Boosting Decision Tree
* rf: Random Forest
* dart: Dropouts meet Multiple Additive Regression Tree

dart might increase accuracy and get process the data faster but the gains in speed might diminish after the model loses the contributing effects of the additional regularization. gbdt is a classic in terms of boosting algorithms and performs well in most situations with respect to predictability. Random forests might already be overkill in the hyperparameter tuning literature since it is already building multiple low correlated trees that is used to build the prediction.

Tip: random forest might be safer if the data set has highly correlated features.

**2.4. objective**  
This defines the loss function. The objective parameter is a string that defines what loss function to use. You can read through the mathematical definition of the loss function in this blog post or the Wikipedia page. A function that does not change is said to be convex. For a convex loss function, gradient descent can (in theory) perform well. There are no local extrema, i.e the loss is flat on the sides albeit steep at the bottom (this can be seen from the contour lines in the images). For a non-convex function, there's also no guarantee for finding the best solution for objective functions. See the Wikipedia page for a list and explanations of all loss functions.

Tip: For a classification objective, log loss will fit the actual training set response much more than a multi-class error will.

Tip: For a regression objective, L2 loss might overfit to the training set and although the residuals on the training set actually reach 0 with perfect predictions, the model may not generalize ideally to new data. Use L1 loss if you require better generalization to new/newer data.

**2.5. metric**  
This defines the function that is used to evaluate the results. This differs from the loss function, where the latter is used to update the parameters in each boosting step. The values of this metric will determine when the tree fitting should stop. A list of the included metric parameters is given on the official GitHub, and you can find this particular description here.

Tip: The R^2 value of a regression model will increase as more trees are added. F-beta values for a classification dataset might increase as more trees are added since the model can learn the 'true' conditional probability of the data set.

**2.6. subsample**  
This specifies the fraction of observations to be selected for each tree. Selection is done by random sampling for each tree, where there is a probability to pick each observation equal to the subsample given.

Tip: If random\_state is set appropriately all boosting trees will be the same, try this by using model.predict(matrix[:3], pred\_leaf=True)  
Tip: Set subsample = 1, use\_missing = True, fill\_na = '999', and categorical\_feature="auto" for binary\_logloss.  
This will cause early stopping as the trees are identical from 1, 2, and 5.

**2.7. num\_leaves**  
This indicates the number of leaves to be generated by each tree in the boosting model. Leaves are the terminal nodes in each tree. num\_leaves can vary from under-fitting to over-fitting the model.

Tip: When the number of leaves is too big the model can cause over-fitting and may take longer to train and potentially predict since each feature may generate a tree and depend on the number of features inputted, each boosted tree might be of size = num\_leaves.

**2.8. boosting**  
This is a HUGE improvement over traditional gradient-boosted decision trees. Since you are only allowed to use 2GB of memory in a Kaggle competition, using up the memory with a d-matrix might be a bad idea. If boosting = gbdt then each boosted tree will be a histogram of the features and these histograms may add up to a dense matrix. If boosting = dart, the single classifier then will de-construct the boosted tree into a leaf-wise decision tree and will prune the leaf-wise decision tree if necessary depending on the files under the dropout group.

Tip: When should I use gbdt?

1. When size=32, depth=8, and dropout\_rate=0, training time for GBDT is about 4 times faster than for DART for the task of ranking 400k products for 5000 customers.
2. DART is requanted to have the same memory requirements as GBDT. Geometrically this makes sense since a dense matrix might be faster if we require hitogram-based leaf-wise trees and weights with an individual that's smaller than the addition of two of the same functions.

Other parameters:

* **seed:** seed value to initialize the random number generator.
* **lambda\_l2**, **feature\_fraction**, and **bagging\_fraction**: reduce over-fitting.
* **min\_child\_samples**: controls overfitting by specifying the minimum samples per leaf node.
* **min\_child\_weight**: An analogous parameter for regression models.
* **max\_depth**: It controls overfitting by specifying how deep the tree should be grown. A number of leaves can be calculated as 2^max\_depth – 1.
* **subsample**: the fraction of the observations sample to be used for training each tree. Low values can cause underfitting, but it prevents overfitting.
* **colsample\_bytree**: specifies the fraction of features to be used for training each tree. High values can cause overfitting.

## For Better Accuracy[ℑ](https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html#for-better-accuracy)

* Use large max\_bin (may be slower)
* Use small learning\_rate with large num\_iterations
* Use large num\_leaves (may cause over-fitting)
* Use bigger training data
* Try dart

## Deal with Over-fitting[ℑ](https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html#deal-with-over-fitting)

* Use small max\_bin
* Use small num\_leaves
* Use min\_data\_in\_leaf and min\_sum\_hessian\_in\_leaf
* Use bagging by set bagging\_fraction and bagging\_freq
* Use feature sub-sampling by set feature\_fraction
* Use bigger training data
* Try lambda\_l1, lambda\_l2 and min\_gain\_to\_split for regularization
* Try max\_depth to avoid growing deep tree
* Try extra\_trees
* Try increasing path\_smooth

# LGB Classifier

def cross\_validation(X,y,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 = lgb.Dataset(X\_train, label=y\_train)

        dvalid = lgb.Dataset(X\_test, label=y\_test)

        model = lgb.train(params = params,

                          train\_set = dtrain,

                          valid\_sets=[dtrain, dvalid],

                          verbose\_eval =False,

                          callbacks = [early\_stopping(100), log\_evaluation(100)])

        y\_preds = np.rint(model.predict(X\_test))

        if xtest is not None:

            test\_preds = model.predict(xtest) / 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(X\_test, num\_iteration=model.best\_iteration)))

    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

optuna.logging.set\_verbosity(optuna.logging.WARNING)

base\_params = { "objective": "binary",

                    "metric": "auc",

                    "verbose": -1,

                    "random\_state" : 42,

                    "force\_col\_wise":True

                    }

def objective\_pr(trial):

    params = {

                "boosting\_type": trial.suggest\_categorical('boosting\_type',['gbdt','dart','rf']),

*# alias reg\_alpha*

                "reg\_alpha": trial.suggest\_float("reg\_alpha", 1e-2, 10.0),

*# alias reg\_lmabda*

                "reg\_lambda": trial.suggest\_float("reg\_lambda", 1e-2, 10.0),

                "learning\_rate": trial.suggest\_float("learning\_rate", 1e-2, 0.9),

*# Num of boosting rounds*

                "n\_estimators": trial.suggest\_int("n\_estimators", 200, 5000, step = 100),

*# increase for better score/will overfit default = 31*

                "num\_leaves": trial.suggest\_int("num\_leaves", 3, 10, step = 1),

*# depth of tree, -1 for full depth*

                "max\_depth": trial.suggest\_int("max\_depth", 3, 5, step = 1),

                "min\_child\_samples": trial.suggest\_int("min\_child\_samples", 20, 200, step = 5),

                'scale\_pos\_weight': trial.suggest\_int('scale\_pos\_weight',2,7, step = 1),

*# alias colsample\_bytree, basically number of columns*

                "feature\_fraction": trial.suggest\_float("feature\_fraction", 0.4, 0.95, step = 0.1),

*# alias subsample, fraction of sample of whole train data*

                "bagging\_fraction": trial.suggest\_float("bagging\_fraction", 0.4, 0.94, step = 0.1),

*# alias subsample\_freq after how many iters get smaple again*

                "bagging\_freq": trial.suggest\_int("bagging\_freq", 3, 10, step = 1)

    }

    params.update(base\_params)

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

    return np.mean(scores)

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

study.optimize(objective\_pr, n\_trials=5)

best\_params = study.best\_params