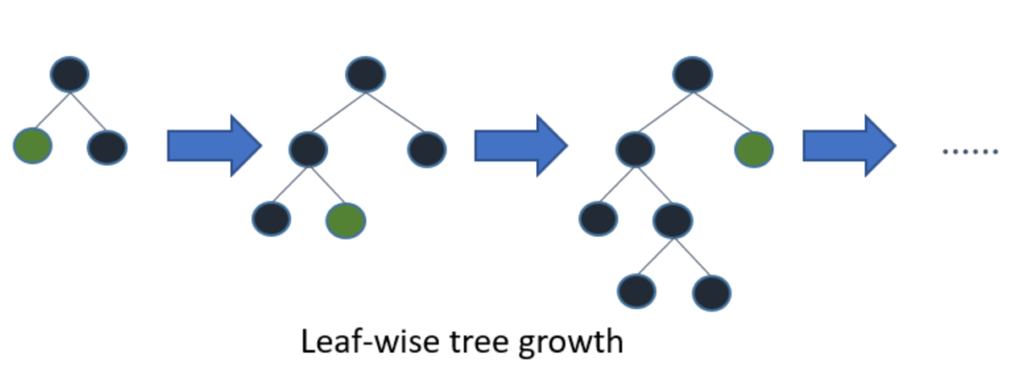
G[radient Boosting Decision Tree](https://analyticsindiamag.com/story-of-gradient-boosting-how-it-evolved-over-years/) (GBDT) is a popular machine learning algorithm. It has quite effective implementations such as [XGBoost](https://analyticsindiamag.com/xgboost-internal-working-to-make-decision-trees-and-deduce-predictions/) as many [optimization techniques](https://analyticsindiamag.com/understanding-adaptive-optimization-techniques-in-deep-learning/) are adopted from this algorithm. However, the efficiency and scalability are still unsatisfactory when there are more features in the data. For this behaviour, the major reason is that each feature needs to scan all the data instances to estimate the information gain of all possible split points, which is very time-consuming.

To tackle this problem, the LightGBM (Gradient Boosting Machine) uses two techniques, namely Gradient-Based One-Side sampling (GOSS) and Exclusive Feature Bundling (EFB). So GOSS excludes the significant portion of data instances with small gradients and only uses the remaining data to estimate the information gain. Since the data instances with large gradients play a more important role in the computation of information gain, GOSS can obtain quite accurate information gain with a relatively much smaller dataset.

With EFB, it bundles mutually exclusive features with nothing, but it rarely takes non zero values simultaneously to reduce the number of features; this results in effective feature elimination without hurting the accuracy of the split point.

Together with these two changes, it accelerates the training time of the algorithm by 20x; with this, LightGBM can be considered gradient boosting trees with the addition of GOSS and EFB.

The LightGBM official document states that it grows the tree vertically while another tree-based learning algorithm grows horizontally; LightGBM grows trees leaf-wise, and it chooses max delta loss to grow. It can be best explained by the following visual.

[Image Source](https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc)

The LightGBM offers advantages like;

* Faster training speed with higher accuracy,
* Lower memory usage,
* Better accuracy than any other boosting algorithm specially handles the overfitting very well when working with a small dataset,
* Compatibility with large datasets, and
* Parallel learning support.

With such features and advantages, LightGBM has become the facto algorithm in the machine learning competition when working with tabular data for both kinds of problems, regression and classification.

Today in this article, we will demonstrate the implementation of LightGBM.

Implementing LightGBM in Python

LightGBM can be installed using Python Package manager pip install lightgbm. LightGBM has its custom API support. Using this support, we are using both Regressor and Classifier algorithms where both models operate in the same way.

The dataset used here comprises the Titanic Passengers data that will be used in our task.

Importing all dependencies

import lightgbm as lgb

import pandas as pd

import numpy as np

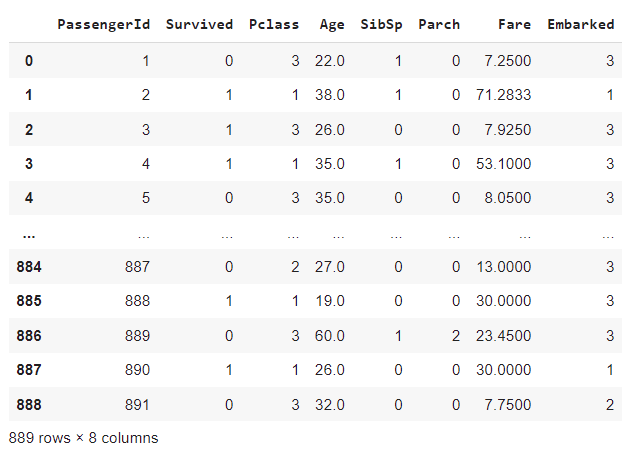
from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

Loading the data:

data = pd.read\_csv('/content/SVMtrain.csv')

data.head()



We have 8 columns out of which PassengerID will be dropped, and Embarked will be chosen as the target variable for the classification problem.

Loading the variables:

# define input and output feature

x = data.drop(['Embarked','PassengerId'],axis=1)

y = data.Embarked

# train test split

x\_train,x\_test,y\_train,y\_test = train\_test\_split(x,y,test\_size=0.33,random\_state=42)

Loading and fitting the model:

The process of Initializing the model is similar to that of normal model initializing; the main thing is that it comes with more parameters settings. While initializing the model we will define the learning rate, max\_depth and random\_state.

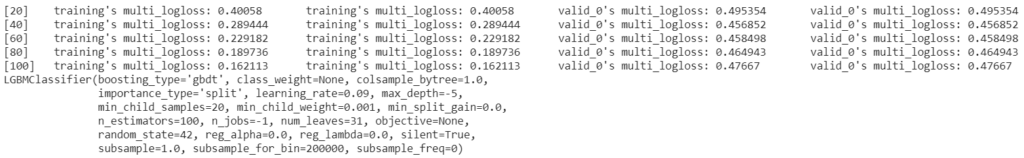
model = lgb.LGBMClassifier(learning\_rate=0.09,max\_depth=-5,random\_state=42)

model.fit(x\_train,y\_train,eval\_set=[(x\_test,y\_test),(x\_train,y\_train)],

          verbose=20,eval\_metric='logloss')

In the fit method, we have passed eval\_set and eval\_metrix to evaluate our model during

training itself.



Evaluating the model:

Our dataset has significantly very low instances; it is better first to check whether this model is overfitted; if not, we will move to further evaluation.

print('Training accuracy {:.4f}'.format(model.score(x\_train,y\_train)))

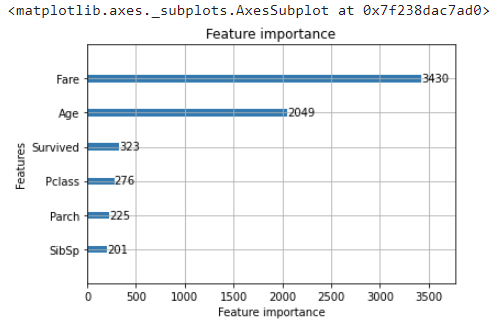
print('Testing accuracy {:.4f}'.format(model.score(x\_test,y\_test)))



As we can see, there is no significant difference between both the accuracy, so the model has estimated the information to be nearly good.

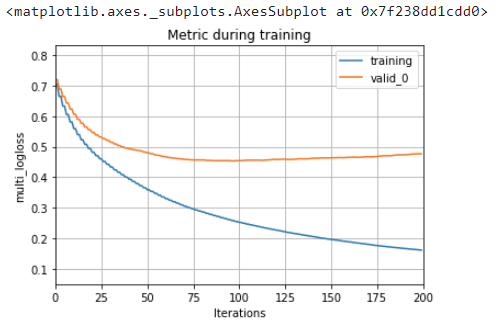
LightGBM comes with additional plotting functionality such as plotting the feature importance, plotting the metric evaluation, and plotting the tree. Below we will see the feature importance and metric evaluation.

lgb.plot\_importance(model)



If you do not mention the eval\_set during the fitment, then you will get the error while plotting the metric evaluation.

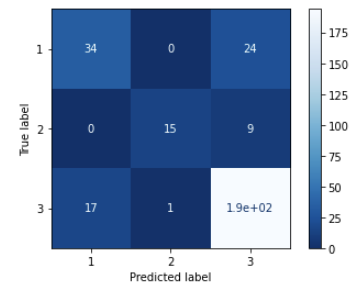
lgb.plot\_metric(model)



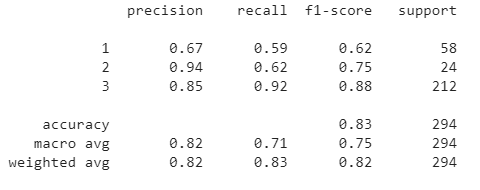
As you can see, the validation curve tends to increase after the 100th iteration; this can be fixed by setting and tuning the hyperparameters under the model.

Below now lets plot the few metrics using sklearn library;

metrics.plot\_confusion\_matrix(model,x\_test,y\_test,cmap='Blues\_r')



print(metrics.classification\_report(y\_test,model.predict(x\_test)))



As we can see from the confusion matrix and classification report, the model struggles when predicting class 1 due to the relatively few instances available for it, but if we compare this result with other ensemble algorithms, LightGBM performs best.  The same procedure is followed for the Regression Problem; we need to change the estimator to LGBMRegressor().