XGBOOST

Sklearn's GBDT implementation is not the best implementation.

The process of hyperparameter tuning / training single model takes too much time.

XGBoost provides optimized implementation of GBDT

- which helps in reducing model training process.

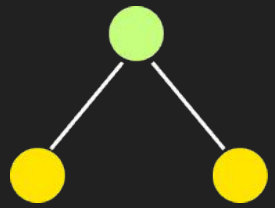
What optimization does XGBoost provides ?

1. Parallelization of features selection

While building Decision Trees,

- there will be n features to consider while splitting the node.

$f_1\ f_2\ f_3 \dots\ f_m$ — M features to consider for split



The computation of Information Gain(s) of the features is done in parallel

- which helps in reducing the training time

$f_1 : IG_1$

$f_2 : IG_2$

$\vdots$:

$f_m : IG_m$

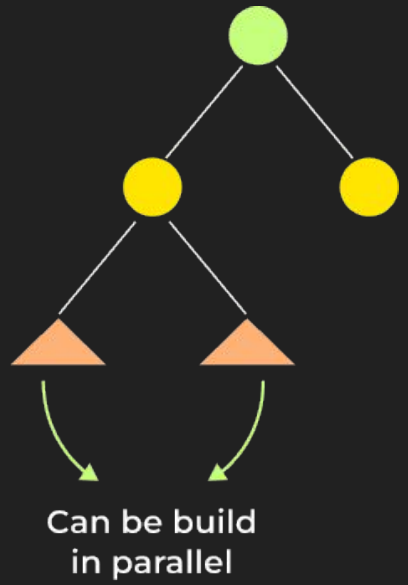
Compute in parallel

2. Parallelization in building DT

While building a DT,

- both subtrees (left and right) can be build in parallel
- as there is no dependency between them.

which helps in making the process faster and efficient.

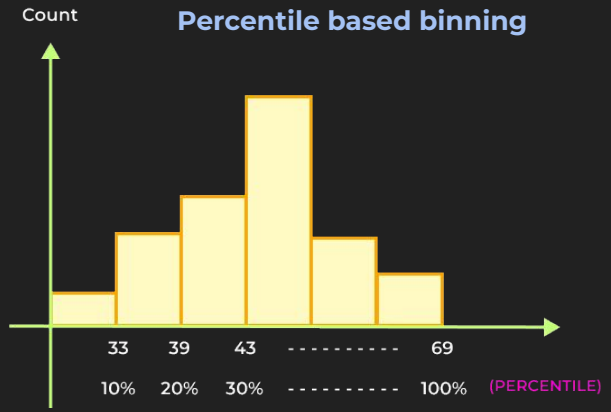


3. Optimizing thresholding in numerical feature

In conventional DTs,

- While finding the threshold for numerical feature
  - all the numerical values are tested to find one with maximum information gain.

SLOW



XGBoost optimizes this by using histogram based binning

- it creates discrete bins (percentile binning) using these continuous values
- then selects threshold using the bins instead of trying every single value.

XGBoost Hyperparameters

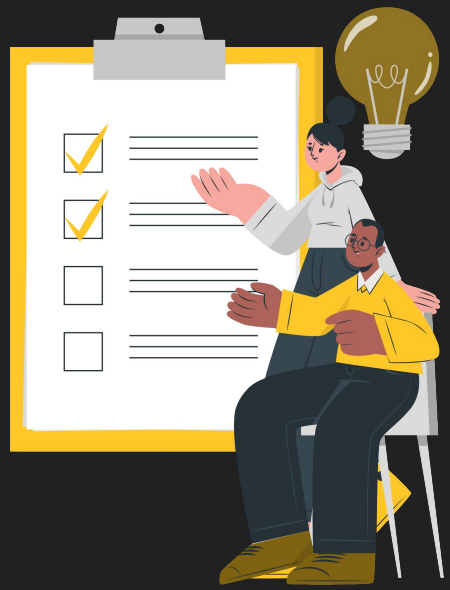
1. **Eta:** or the learning rate is the shrinking/ regularization term
2. **Min_split_loss:** specify the minimum Information Gain which you want for further split.
    - the splitting stops if the min_split_loss is not met.
3. **Max_depth:** set the depth of the base learners
4. **Subsample:** row sampling rate.

- **colsample_bytree** (*Optional[float]*) – Subsample ratio of columns when constructing each tree.
- **colsample_bylevel** (*Optional[float]*) – Subsample ratio of columns for each level.
- **colsample_bynode** (*Optional[float]*) – Subsample ratio of columns for each split.
- **reg_alpha** (*Optional[float]*) – L1 regularization term on weights (xgb's alpha).
- **reg_lambda** (*Optional[float]*) – L2 regularization term on weights (xgb's lambda).

It provides various levels of column sampling

- for each tree
- for each level
- for each split

Also, it provides L1 and L2 regularization parameters



LightGBM

- published in 2017 by Microsoft Research

LightGBM is another implementation of GBDT

- uses insane optimization to make the training efficient.

Surprisingly, it is faster than **XGBoost**

What makes LightGBM faster ?

1. **GOSS (Gradient-based One-Side Sampling)**

Say, we are training the m^{th} model,

- During training, there will be lot of data points with small residual

LightGBM will drop all the points from training where the error is very small.

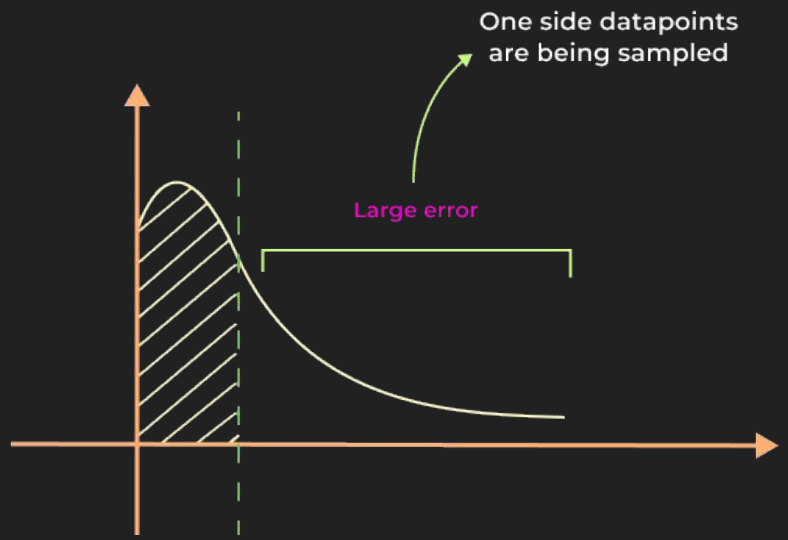
Intuitively,

- It is doing **smart sampling** by
  - reducing the size of training data
  - which make the training process faster.

Why does one side sampling means ?

If we were to plot the error distribution,

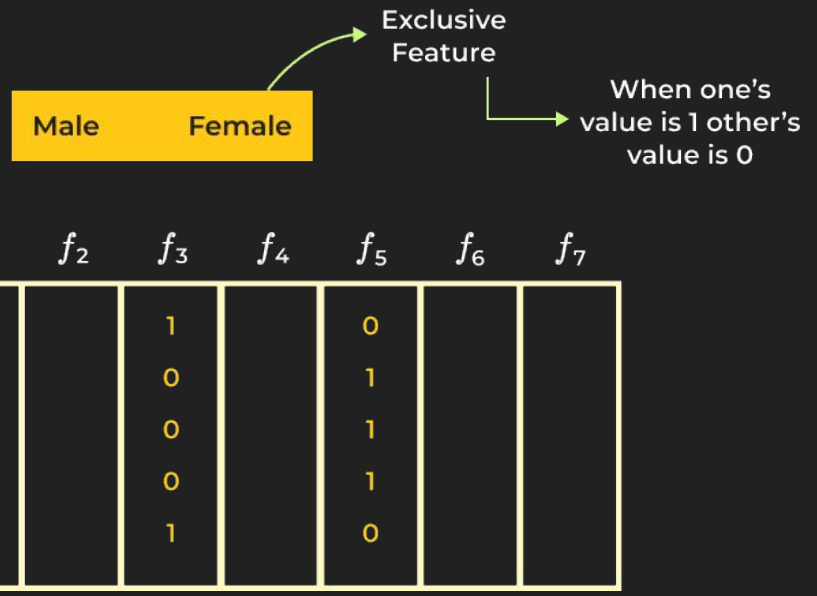
- We are sampling the points from one side of the distribution



2. EFB (Exclusive Feature Bundling)

It scans through all the features

- tries to find feature pairs which are exclusive



What does exclusive feature mean ?

Say, we have an OHE encoded categorical feature

For example:

- **Male & Female.**

What happens after Exclusive feature pair is identified ?

It'll group the features into single feature

- and create a new feature

Such that new feature will have information of both the feature

Intuitively,

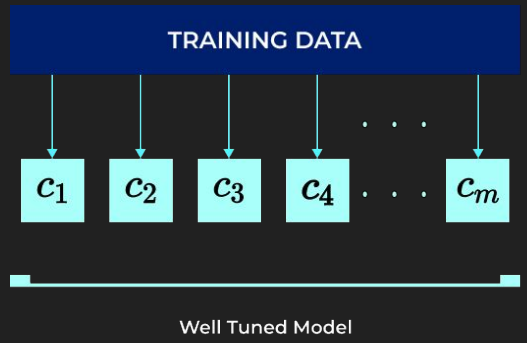
- it is performing **dimensionality reduction**
  - which helps in training GBDT faster.

Stacking

How does stacking works ?

Let's say we are entering a kaggle competition

- **and we have a team of m members**



The team decided that each individual member will train its own model.

So, on a give training dataset - there will be m well hyperparameter tuned model

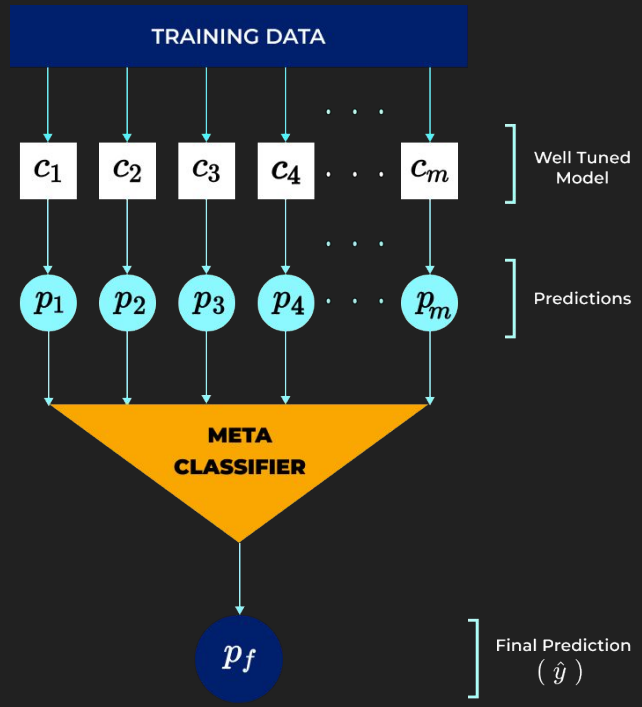
Note that

- all m models can be different
- i.e. C_1 can be logistic regression, C_2 can be Knn etc

How do we combine them?

Each model will give us a prediction (P_1, P_2, ... P_n)

- We train another model (**Meta Classifier**)
  - using predictions as input data
  - and original target variable (y) as target variable



Note : Instead of predictions, we can use class probabilities as input feature for meta classifier

The prediction given by Meta Classifier is treated as final prediction

why don't we use it ?

- **extensive time complexity**

It takes a lot of time to fine tune M base models

- and then training the predictions on Meta classifier

NOTE :

- Stacking is extensively used in kaggle competition as kaggle competition goal is to get the top score
- and not the fast models

CASCADING

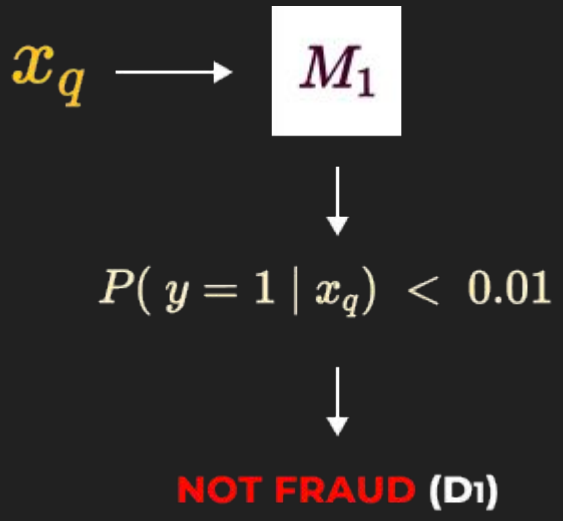
Chaining of models

Let D = dataset [transaction fraud or not]

- Y = 1 (fraud)
- Y = 0 (not fraud)

For a query point

- we will pass this point through the first model M_1
- Model M_1 will return the probability of the query point being a fraud



Based on probability, we'll split it in 2 parts:

- if the probability of $\hat{y}$ being 1 is extremely low, say < 0.01 then
    - we consider that as not fraudulent, let this data be D_1

What happens to rest of the data?

The rest of the points ($D - D_1$) i.e. data with prob. > 0.01 which we are not sure about

- will be passed through the next model M_2
- Model M_2 will be more stricter i.e. it'll penalize more.

Again model M_2 will split into 2 parts

- non fraud (say, $P(y=1|x_q)<0.001$)
- fraud transac. ($p > 0.001$)

We keep doing this till n models

Training Data : D