Table of Contents

[1. Feature Engineering: 3](#_Toc194042549)

[1.1. **Feature transformation:** 3](#_Toc194042550)

[1.1.1. Outlier Detection: 3](#_Toc194042551)

[**1.1.1.1. Z-Score method:** 3](#_Toc194042552)

[1.1.2. Missing value Imputation: 4](#_Toc194042553)

[**1.1.2.1. KNN imputation:** 4](#_Toc194042554)

[**3. Supervised Learning:** 12](#_Toc194042555)

[**3.1. Linear Regression:** 12](#_Toc194042556)

[**3.2. Logistic Regression** 12](#_Toc194042557)

[**3.3. SVM: Support Vector Machines.** 14](#_Toc194042558)

[**3.4. KNN.** 15](#_Toc194042559)

[**3.5. Naïve Bayes:** 15](#_Toc194042560)

[**3.6. Decision Tree:** 16](#_Toc194042561)

[**3.7. Ensemble Learning:** 24](#_Toc194042562)

[3.7.1. Voting: 24](#_Toc194042563)

[3.7.2. Stacking: 25](#_Toc194042564)

[3.7.3. Bagging. (Bootstrap + Aggregation) 26](#_Toc194042565)

[3.7.4. Boosting: 28](#_Toc194042566)

[**How boosting works:** 29](#_Toc194042567)

[**Gradient boosting algorithms:** 29](#_Toc194042568)

[**For classification (Difference specific to classification):** 32](#_Toc194042569)

[**Why gradient boosting is a gradient descent in a function space?** 32](#_Toc194042570)

[3.7.5. XGBOOST: 33](#_Toc194042571)

[**Core idea of XGBOOST:** 33](#_Toc194042572)

[**Mathematical foundation:** 33](#_Toc194042573)

[3.7.6. LIGHTGBM 35](#_Toc194042574)

[**Leaf base split:** 35](#_Toc194042575)

[**Histogram base learning:** 36](#_Toc194042576)

[**Categorical data handling:** 36](#_Toc194042577)

[**Exclusive feature bundling:** 36](#_Toc194042578)

[**GOSS (Gradient base one side sampling):** 37](#_Toc194042579)

Bias vs variance:

Learn it from a book and lecture in way to depth with numerical calculation.

Bias: Bias is the error introduced in the model’s prediction due to over simplistic nature of the model. It reflects / measures how much the models’ predicted outputs deviate from the true values, often because model is unable to capture the full complexity and patterns of the data.

Bias the words

**Why is bias called bias?**

Bias indicates the bias nature of the model to predict values towards its ideal simplistic nature, like linear regression is bias towards linear relationships irrespective of the real nature of the data. Naïve bayes is bias towards the feature independency. This is the start of the origin of the concept, bias indicates how good model can fit on intricate patterns with no restrictive assumption. High bias means high inclination of the model towards the restrictive assumption, very simplistic model like linear regression. High bias means models simple model which makes the mistakes on the training data itself. Whereas the low bias means model does not incline towards the specific restrictive assumption.

Sources:

Choice of the algorithms, like selecting linear regression for the highly non-linear data.

Important relevant features are omitted.

Non-representative training data,

The best geometrical intuition is you fit a linear model on the non-linear data.

Quantify Bias?

* Quantifying bias is based of the performance of the model on the testing set.
* The overall error is bias \*\*2 + variance + irreducible error.
* Quantifying bias here?

Why do we need overfitting and underfitting if we have bias and variance concept.

* Bias is the mathematical statistical concept which calculates the actual error/deviation in the model prediction due to oversimplistic nature of the model.
* Whereas the overfitting is the reason why bias happens in the model.

Example: for house prediction you only select only location as one input variable and then make the prediction base off the location, which makes the prediction very biased towards the location variable only, though the actual dependency is on many variables.

Why do we need bias and

Variance

Trade off

Overfitting

Underfitting

Overfitting is like low bias.

# Feature Engineering:

## **Feature transformation:**

Transforming the raw features into a format which better represents the data pattern. It includes changing the existing features.

### Outlier Detection:

You either removes the outlier values based on your detection techniques or you treat them as missing and then try to fill them or you can cap those values to the extreme allowable values based on your model requirements. Or you can do the power transformation before that might just handle the outlier. Most importantly the tree-based algorithms do not really get impacted by presence of the outliers.

#### **Z-Score method:**

Mainly used on the normal distributed data where, we calculate the Z score of each point which indicates how far away the point is from mean in terms of the standard deviations. Or rather how many standard deviations away the point is from the mean. And then standard deviation in our case is the unit of the variability in the data, so basically indicates how much is the variability of the point around the mean. i.e. you calculate the z-score and basically the variability of the points around the mean and if this variability is beyond a certain threshold like 3 or 3.5 which basically contains most of the data, i.e. value of variability beyond these points means it lies beyond majority of the data and can be considered as outlier.

So just calculate the z-score and take only the values which are within the threshold. S.D. = 3 is general approach.

Data needs to be gaussian to use Z-score method of outlier detection. And if it is not then in that case a **modified a Z-score method** is used which uses median and IQR with the scaling multiplier to reduce the value of this z score to make it more comparable with mean z-score as median z-score values come up as bit bloated, so to have same comparable threshold we reduce those z-score values from the median modified z-score. Uses MAD, mean absolute deviation. Modified Z-score method is generally a better approach than IQR.

Mean, mode, median values remain same for normal distribution data, whereas for skewed it changes, mode obviously is to the extreme and it remains towards the extreme. For left skewed data i.e. tail on the left sequence goes like this, mean, median, mode. And for the right it goes like mode, median, mean.

The way to understand it is, first assume the normal distribution and then add the outliers at the ends and try and see how it changes the mean and median, adding the outliers towards the right making it right skew, this will increase the mean by very significant amount but won’t change or increase the median by significant amount as however much could be the extent of this

* + - 1. **IQR method:**

In this method a IQR is used to detect the outliers in the non-normal data which uses IQR percentile approach, the lower and upper limits are decided by the Q1-1.5(IQR) and Q3 + 1.5(IQR) respectively.

* + - 1. **DBSCAN:**

implement in notebook and understand.

### Missing value Imputation:

For numerical feature, make sure you try to keep more than 95% data if you are going for CCA.

And on imputation the distribution of the numerical variables should not change too much, Check for KDE distribution.

For categorical with regression problem, see the distribution of the output target variable with respect to that categorical variable before and after the imputation and make sure that distribution of those all the categorical variables are not completely changed.

Dig deep into this thing, like how it changes and if it changes what should you change.

#### **KNN imputation:**

Based on the K nearest neighbours, but the nearest value is calculated based on the variables/values which are available, just ignore the dimensions where the values are null. And weight is used in calculating the output. Distance as a weight or any custom weight or just normal without any weight.

A weight is reciprocal of the distance is used as the distance is inversely proportionate to the impact that point have on the imputation value.

By ignoring the dimensions which are not available you basically do not calculate the actual distance, but you calculate the planer distance as you do not have the exact value of that dimension. But as you do the same for all the points it evens out the error as ultimately you calculate the imputed value based on the points aggregation not the distance aggregation.

Not specifically used for categorical data. So does the iterative imputer.

**Missing completely at random, missing at random, missing not at random.**

**After the encoding of the categorical features, scaling is not done on those features.**

**Multivariate Imputer:**

The columns which are filled at the last are much better filled than the ones which are filled initially.

* Initially the values are initialized with mean etc.
* For the first iteration the first column null value rows are treated as test data and the remaining data is treated as train data which is trained with first column treated as target column. The prediction is done for the test part, this prediction is nothing but the imputation of the first column which based on the regression model.
* For each subsequent column you use the newly updated values and not the values from the earlier iteration.
* Likewise, you do for the all the subsequent columns, for each current column you use the better and better data for the model as the previous values are filled based on the regression and not just the mean initialization which is the case for the earlier columns.
* The intuition is that we try to capture the numerical patterns between the features and use those patterns to fill the imputed data. But pattern capturing is done iteratively as model starts with the initial mean values which with each iteration improves because it gets more and more accurate data for next feature values prediction / filling.
* Max iterations are set based on the subsequent changes in the next iteration.

Main question is how do you used it on the mixed data? i.e. data with numerical and categorical features as well.

**Column Transformer and Pipelines:**

Use to streamline the preprocessing and prediction steps on the training and testing data. Very easy to optimize way of preprocessing and prediction with very much ease of automation, particularly useful for production purpose. Although pipelines work seamlessly with cross-validation as well, so very useful in model building stage as well and not just the production automation.

Column transformers transforms the columns one by one as it given to it in its steps and joins those columns alongside and returns a NumPy array. Column Transformers works parallelly combining all the data points.

When you specify a column twice in a distinct transformation then even the second time it receives the initial column only and adds it along with the other column and now, we’ve two transformed columns out of single initial column. The natural intuition of yours is that for the second transformation you’ll received transformed column after first transformation is not true.

You cannot apply two distinct column transformers on the data in series as the first one returns the NumPy array, so the second one receives the NumPy array and fails because of it. The solution is you use pipeline in column transformers, which allows the serial transformation on a single column in transformer. For a one transformation within a column transformer use pipeline and specify multiple serial operations.

The intuition is you pick one by one column and with the help of the pipelines specify all the serial transformation for that column in one transformation of the column transformer and likewise do it for all the columns and eventually you’ll get the NumPy array which you can fed to predictor without any issue.

Whereas the **pipeline** works serially on the different transformation’s steps put into to the pipeline, i.e. the first step will be processed first on entire data and then the second and then the third obviously how and what is depends on which step and what methods is used.

The last step in pipeline is by default treated as the predictor and works slightly different than all the previous steps.

All the transformer steps on given fit which is one of the most basics of the method does **fit\_transform** on the data whereas it does only fit on the predictor. And we do fit on the train data, i.e. for training by fit method of pipeline we do fit\_transform on transformation and only fit on the predictor which is the last step.

Likewise for the test, we do predict, which does the only transform on the data, which is test data here, and predict for the predictor step on the test data usually.

Pipeline does not do any transformation, column transformers and predictor do the transformation, but pipelines just combine those transformation in serial manner, outside of column transformers or within the column transformers.

Just like pipeline there is one more function make\_pipeline which is simpler version of the pipeline which receives the list as a parameter with each element being the estimator object directly unlike the Pipeline which receives the list of the tuples where first element of the tuple is name of the estimator and second element being the object of the estimator. The name to the estimator step is taken as the lowercase name of the estimator itself.

Major advantage of the pipeline over the make\_pipeline is its ability give custom meaningful names. Though it provides more flexibility and control in certain complex scenarios.

* + 1. **Encoding:**
    2. **Feature scaling:**
* Feature scaling is a technique is used represent the numerical data into a common scale so that larger values do not dominate the model predictions. Makes sure all the features contribute equally. Models which depend on distance metric particularly, a feature with higher value dominates the distance and eventually model’s prediction, to avoid the same we get those values in similar scale keeping the distance/distribution within the feature same.
* Very useful for distance-based algorithms like KNN, Linear models, KMeans, SVM and the models which are based gradient descent like NN which converge better with scaled features as the normalized features coefficients are kind of similar unlike very much difference in them cause much smooth transverse towards the optima rather than messy zig-zag movement towards the minima.
* Not necessary for tree base models, as these techniques does not change the variance within the features and variance within features is not dependent on the scales of features as well.
* There are two methods to do it:
  1. Standard Scaler: Use Z-Score method to change the scale of the data. What Z-Score does, it basically get’s the data into a scale where the mean of the data is 0 with S.D. is 1, keeping the distribution same, i.e. **shape remains the same, just the scale changes such a that it shrinks in a way where all values get distributed around zero mean with S.D. as 1. It is a linear transformation which does not change the shape just shrinks the shape, to change the shape it must do non-linear transformation like log, power, yea-Johnson transformations.**

More robust to outliers compared to the normalization as normalization depends on the range which very much depends on the max and min values.

Circles the values around 0 as mean.

* 1. Normalization: changes the values in a fixed range unlike the standardization. (0-1,-1-1) etc.

Min-Max

Check from notes and fill it next time when you revise.

* + 1. **Power transformations:**

**First get clear with homoscedasticity and then come to this topic.**

**Let's go littel deeper into the thing, let me tell my understanding of the constant variene or homoscedacity, let's say you have 100 datapoints and you calculate the varience for it, it comes out to be some number, now obviously this number is average of the squared distances of those points from the mean, so it basically averge out the individual datapoint distance / varience with respect to the mean. now this individual distances could be very high for some datapoints and could be very very low for some datapoints, and this will eventually be balanced out and we'll get a single value, which represents the average sort differenes/distances/varaition of values with respect to the mean. now if the if the data does not have this variation of some datapoints distances high and some's low, i.e. alomsot every datapoint has near about same distance/squared difference then we can it is constant varience of homoscedacity is there and viceversa for hetroscedacity, is this understanding correct ?**

Data Leakage:

* How to check?
* To check for data leakage, shuffle your dataset, split it into train and test sets, train your model, and repeat this 5–10 times, noting test accuracies. Stable, high accuracies (e.g., 89%–92%) suggest no leakage if preprocessing is clean. Large variations (e.g., 85%–95%) or a drop with stricter splits (e.g., time-based, falling from 90% to 75%) indicate possible leakage. Compare with training accuracies and audit your pipeline—ensuring preprocessing and features don’t use test data—to confirm. This flags leakage efficiently, ensuring accurate model assessment.

1. **Metrics:**

**MAE and RMSE:** what is the difference between them on intuitive level, just imagine the bars for each data point bars are just to one side or some part to left and some part to the right, then averaged them if the bars were an absolute difference which will give you an another bar which is nothing but the crux and aggregation of all the bars.

In case of the MSE and RMSE the bars are elongated by quadratic proportionate, which gives more weightage to the outlier points with the degree of two. To get the bars in same unit as the base data we take root of the MSE which is the average of the squared differences. i.e. MSE is the average of the squared bars whereas the RMSE is the root of the mean of the squared differences and not the mean of the rooted differences.

The RMSE can be calculated by two methods, first the correct approach and other one is rooting the sum and then dividing by the N, which is wrong but why?

The reasoning is intuitive level, Imagine the bars and on introduction of the square terms, the bars are elongated by the square proportionate, now our error is this value of each point, and we want to average it for all the points so that we can have its representation for each point. The average of this point is calculated by only one method which is sum the difference, which is the sum of the squared distances, which in this case represents the total error and for average we need to have the total distance or difference and just take partial value of it for each datapoint by dividing by the no. of datapoints. Which represents the squared error for each datapoint and then we take the square root of it to get the error representation for each datapoint in actual units. And if do the square root of the total error first then it does not remain the total error to distribute it to the each datapoint by dividing N. It gives a smaller value which is the underrepresentation of the actual average error.

MAE is more robust to the outliers whereas the MSE AND RMSE penalizes them much more.

If the data is suspected to have the outliers, then use RMSE or MSE depending on the context obviously.

Theory in details with every little nuance including all the hyperparameters along with the code implementation of the everything.

**R2 SCORE (Coefficient of Determination):**

So, basically the R2 score the competitive model which calculates the how well the current regression model fits with respect the mean model, so it checks the relative sum of squares i.e. sum of squares of all the datapoints by regression model with respect to the sum of square of all the datapoints by mean model. just to get the values in the specific range for all the comparative model it checks the error with respect to sort of one absolute model which is mean model, so that we can get a standard value between the 0-1 and for the models across different model.

The R² score is a comparative measure that evaluates how well your regression model performs relative to a simple mean model.

R2 scores allows the comparisons between the models based on how much variance in the data is explained by each model. If the r2 score is 0.7 then the 70 percent variance in the dependent variable is explained by that specific model which obviously translates to the other errors as well.

R2 score does not necessarily inversely proportional to the MAE

If you develop **Model B** that fits well to the non-outlier data points, it can reduce the sum of squares for those points and improve the R² score from 0.88 to 0.90. However, this model might move further away from the outliers, leading to larger absolute errors for those outliers and thus increasing the MAE from 0.50 to 0.60. This illustrates that while R² measures the proportion of variance explained by the model, MAE is more sensitive to individual large errors, especially those from outliers. Both metrics provide valuable but different insights into model performance. In this scenario, **Model B** shows an improved R² score by better capturing the variance for non-outlier data points, but its MAE increases due to the larger errors associated with the outliers.

R2 score is not the correct metrics in the scenarios where:

1. Significant outliers
2. High no. of features, particularly the non-useful
3. Small sample size
4. Non-linear datasets are not that straightforward to interpret.

the adjusted R2 score is used cause on increasing the no. of feature the model complexity is increase which increases the R2 score but to make sure the increase R2 score is justifiable that is due to the actual feature which add the understanding of the variance in the dependent variable and not just the increased in overfitting we use adjusted R2 score which increase or decreases the R2 score with respect to the no. of independent variables, how does it balances, increases, decrease or changes in general R2 score. It balances the R2 score with respect to the number of the independent feature.

MAPE (Mean absolute Percentage Error): same as MAE but just the relative of the MAE. Should be as low as possible.

Mean deviation Bias (MDB): summation of the errors without the mod, tells you about the over, under or exact prediction tendency of the model.

MDB > 0 = Model under predicts

MDB < 0 = Model over predicts

MDB = 0 Model is not biased towards any side.

Classification metrics:

1. Accuracy: total right to the total datapoints.

Accuracy is very vague way to judge a metric, and not the best metric in lot of scenarios:

* Imbalanced data: high accuracy score in the imbalanced data can be due to the correct classification major class and minor class accuracy could be very low.
* It does not distinguish between the type of the errors: In some scenarios few errors could be very costly compared to the others. Like in medical data where you are predicting cancer, False negatives are very costly compared to the False positives.
* Types of the errors in the Classification Metrics which are a bit different compared to the Type of errors in hypothesis testing.

Type 1 error: False Positive (Precision, critical in email spam)

Type 2 error: False Negative (Recall, critical in Cancer)

Type of errors in the hypothesis testing:

* Type 1 Error: reject the null hypothesis when it’s true.
* Type 2 error: accepts the null hypothesis when it is False.
* One true thing about the metrics is that they are not the absolute evaluation criterions for the models, you mainly try to check if the model is correctly doing the particular aspect which you want the model to do correctly. Most evidently it is seen in the Recall, precision in below metrics.

1. Recall / (Sensitivity) / True positive Rate (TPR):

When the cost of predicting the Positive class as negative is much more detrimental compared to the other way around then in that case we recall.

It mainly focuses on the fact that make sure you predict all the positive classes as positive and the other class’s correct prediction is not that important.

In recall you try to calculate the true positive with respect to the all the positive classes in the dataset.

i.e. Recall = TP / (TP + FN) 🡪TP / horizontal (Where horizontal is actual)

in Recall you do not give any fuck to the negative class. If only the positive classes are classified correctly out of the entire data, then also we can Recall as 100% or

rather if we classify all the datapoint as the positive the recall still would be 100 but that would be bad model.

Only using the Recall is probably not the correct strategy for the model evaluation, as recall value can be misleading in lot of sense. It is very easy to have high recall with classifying all the points as positive. As well it does not considers the FPR.

1. Precision:

When the cost of the predicting the negative class significant then in that case it important check the how many of the negative classes are predicted incorrectly with the correct prediction of the positive class.

Predicting the positive class correct is always the goal but with that you also want to make sure the negative class is not misclassified, then in that case you use the Precision. We want to make sure, out of the predicted positive values the falsely predicted values very less as the cost of those misclassification is much more, though the goal is predicting positive class which is not achieved by the precision for the same we can use recall. And that is why the Precision and Recall are used together or in combination.

It’s just that we want to predict the correct positive class, but we do not want to compensate for the negative class’s false prediction which was not that important in the initial context of precision.

Recall is the indicator of the correct prediction of the entire positive class, and tells me nothing about the other's class's misclassification, whereas the intuition for the

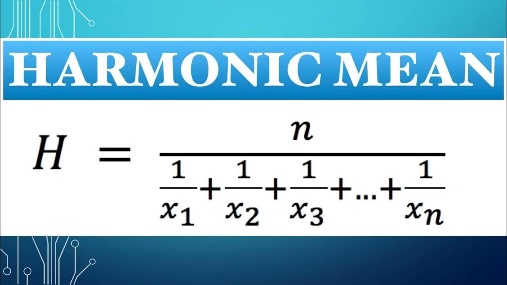
Precision is, when model predicts high precision it indicates, out of the positively predicted classes by the model most of them are true positive only and the misclassification of the negative class is not that much which is more detrimental in the above scenarios.

Recall focuses on the correctly predicting the positive class whereas the precision focuses on the not including the negative class in the positive prediction. Not it mainly focuses on or ensures that the model does not predict the negative class as the positive. It has no sense of how many of the positive classes are predicted correctly all it tries to check that it does not have any negative predicted as positive.

Positive class: The class of the primary interest.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Predicted** | |
| **1** | **0** |
| **Actual** | **1** | **TP** | **FN** |
| **0** | **FP** | **TN** |

Harmonic Mean:



**F1 score** aggregates both the scores based on harmonic mean where the beta values are 1.

The more underlying metric is F-Beta score.

Beta gives you the ability to adjust the impact precision and recall in calculation of the F-Beta score.

β < 1: More weight to precision

β > 1: More weight to recall

β = 1: Equal weight (equivalent to F1 score)

chipake khichta hai neeche.

Weighted F1 score is better at handling imbalance than the normal F1 score, which is the weightage average of the F1 score with respect to positive class and F1 score with respect to the negative class.

ROC curve (Receiver operating curve):

A curve plotted between the TPR and FPR for a model with different threshold.

First, what is TPR and FPR?

TPR is nothing but the Recall/Sensitivity.

FPR is (FP / FP + TN) i.e. with respect to the negative class how many of those are misclassified, means more the misclassification of the negative class more will be the FPR, so you want it less whereas the TPR is you want it as high as possible as it tells you how many of your positive classes are classified as positive.

FPR and TPR are noting but the indicators that your model is good at predicting the positive and negative values.

Plotting a ROC curve for a model, you take different threshold and calculates FPR and TPR for each threshold for that specific model and plot those values for each threshold you get a curve.

High AUC-ROC value indicates the model generally across the thresholds have better ability to distinguishes between the positive and negative classes, cause the high AUC means high TPR which means better ability to classify the positive classes.

At the same time high AUC indicates the low FPR which implies the low FP’s which implies better ability of the model to identify negative class which also implies the better ability precision.

High AUC-ROC value, better ability of the model to distinguish between the positive and negative classes with high Recall and High Precision.

AUC curve which travels closes the top left corner is selected means that model as it has more area (Usually).

# **Supervised Learning:**

# **Linear Regression:**

## **Logistic Regression**

Logistic Regression: The basic intuition which is to separate the classes by a linear plane based on the distance (probabilities which are proportional distance) to the i.e. Furthest the point is from the model plane better predicted that point is. But how do you get the model. Random model improved by the gradient descent.

Once you’ve a model with respect to that linear model, you calculate the distance of the datapoints from the linear model, furthest the point, better classified it is. In general, the model which keeps the most distance from the model and datapoints is the best model. So, once calculating the distance of all the points we just product the distances of all the datapoints irrespective of the classes and whichever model gives you the most product of the distance is selected. (Obviously you can use step function which would just tell you which point is classified correctly, and which one is not but it does not value the higher chance of point to be classified as certain class). But generally, this probabilities are very small and product of this is also very small so, you basically add log into it which makes everything negative you instead of wanting it higher you want the value of the product minimum.

This log addition of the probabilities we call as likelihood. From the same likelihood we get the concept of the log loss error which is binary cross entropy which is smart logical way to calculate the probability of each datapoint with respect to each class.

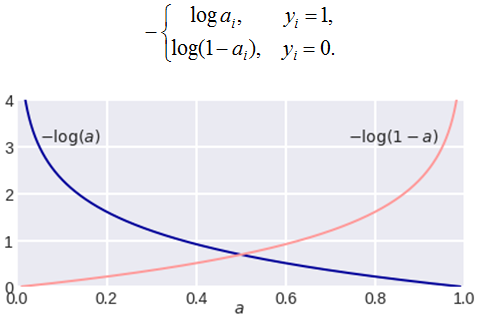
Actual step:

* Calculate the distance.
* Probabilities from the distance by the Sigmoid / SoftMax.
* Error from the probabilities by log loss error.
* Based on the error, by the gradient descent calculate the model and do it iteratively, obviously the error has the feature variables because it based on the probabilities, as probabilities has Z which is the distance marker which has the feature variable which ultimately represent the model. i.e. means with gradient descent, derivative of the loss with respect to the feature variable is sort of very indirectly dependent on the loss.

Loss function: Log loss Error / Binary Cross Entropy:

It is designed such a way that for the positive points only the positive values error is calculated and vice-versa.

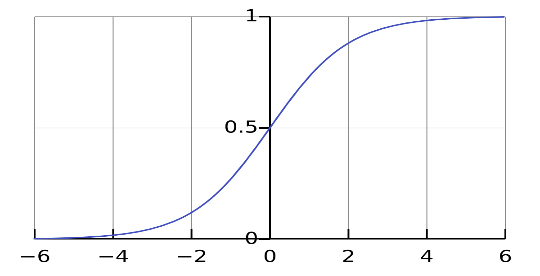
And obviously the log is for the handling the small values and is derived from the



Function in the logistic regression:

**Sigmoid:** The functions “S” shaped curve which is convert the minus infinity to positive infinity values into the range of the 0-1. The higher the X higher the output is more leaning towards the 1. So there definitely the positive correlation between x and y here, just the exponential one. Symmetrical along the y axis. And at the x= 0 the output is 0.5.

Specifically used to convert the distances in the logistic regression into the probabilities.

****

**SoftMax:** a function which returns a exponential normalization of an array given. Which in out context basically represents a relative probability of classes. So called as uncalibrated probabilities as they do not have any absolute sense, just the relative sense which gives more weightage to the higher values.

How it handles the multiclass problem?

In general approach in multiclass problem is to create as many hyperplanes as the number of the classes. And now there are two different ways to create this hyperplane first OVR and the second one is SoftMax.

There are two methods.

* + 1. **OVR (one vs rest).**
* You one hot encodes the target variable and for N number of classes N number of the models are created with respect to each class. And the normalized probabilities are calculated for each datapoints of the all the probabilities given by each model as each model return the probabilities in the scale of the 1, so we need to convert those into scale of 1. And whichever probability comes the highest, the datapoint belongs to that class.
* So, basically multiple models are created and trained each has its hyperplane for binary classification unlike SoftMax which handles multiple planes in a single model.
* In OVR, for each model/hyperplane a probability of the 1/positive of that class (a probability of point belonging to that class) is calculated which is then normalized.
  + 1. **SoftMax:**
* In SoftMax multiple hyperplanes created and handle within a single model with the inclusion of the SoftMax function which handles the multiple probabilities in a single model.
* A distance (Z) for a single datapoint is calculated for all the N classes with respect to their hyperplanes. And a SoftMax return the relative probabilities of the classes out of which the highest one is selected. What SoftMax does is? it calculates the normalized exponential of the distances (Z) of each class with respect to their hyperplanes. And this normalized exponential value is nothing but the relative probability of the classes (or rather relative probabilities of the distances which are great indicator of class classification or geometrically a region indicator of that point).
* And this max value decides to which class this point belongs.

Just the main difference is in the way the SoftMax and OVR integrates the multiple hyperplanes, OVR creates the multiple models and select the highest normalized class out of all whereas the SoftMax can handle multiple hyperplanes in single model returns the highest relative probability of a point to belong to a class or region.

Number of hyperplanes:

Softmax: Uses K hyperplanes for K classes.

OvR: Also uses K hyperplanes for K classes.

Training approach:

Softmax: Trains all hyperplanes simultaneously

OvR: Trains each hyperplane independently

Decision making:

Softmax: Uses all hyperplanes together, applying SoftMax function.

OvR: Considers each hyperplane separately, often using sigmoid function.

Output interpretation:

Softmax: Directly outputs probabilities that sum to 1

OvR: Outputs independent probabilities for each class

Computational efficiency:

Softmax: Generally, more efficient, especially for many classes

OvR: Can be less efficient, especially with many classes.

So, while both methods use multiple hyperplanes, their approach to training and using these hyperplanes differs significantly.

Interview Questions on the logistic regression:

Why logistic regression is called logistic regression?

First the logistic part comes from the logistic function which is nothing but the sigmoid function. Logistic probably comes from the logistic number which probably comes from the logarithms.

The regression part is because it uses linear combinations of the features to calculate the probability of the datapoint and ultimately calculates the linear plane which distinguishes the classes. Cause the regression was used in linear Regression and due to their historical relevance, they have similarities in a way they represent the features.

## **SVM: Support Vector Machines.**

Support vector machines

At the end make sure to add the similarities and differences between the SVM and Logistic Regression.

## **KNN.**

## **Naïve Bayes:**

Uses the probabilistic approach by bayes theorem to decide the highest probably of a target class given the input conditions which is input features combinations. So, technically based on the input features combinations the probability of the all the classes is calculated whichever has the highest probability of occurrence for the given condition or given combination of the features is chosen as the correct classification.

The point is how do you calculate these probabilities. Bayes theorem is used to calculate the conditional probabilities, what is the probability of the given class given the condition.

The proof is as such, to calculate the probabilities of classes given the condition, with the bayes theorem for the above probabilities we calculate the probability of the conditions given the class and multiply by the prior probability of that class. So, the main issue lies in the fact how do you calculate this first term of probabilities of the feature occurrence given the class, it turns out that this terms simplifies to the individual probabilities of feature values for that datapoint for that class, which is fairly easy to calculate now, as we just sum this category value for the given class and divide by the total number of the class occurrence, basically calculate for a given class and given category how many of this class occurrence has this probability and you do the same for all the classes and do it for all the categories of a feature then repeat for all the features.

Learning in algorithm is basically just the probability calculations and stored in dictionary and on prediction, you just call the value puts it into a model calculations whichever class is more likely is our output.

Naïve bayes works good with the small datasets.

For the numerical features you use probability distribution, mostly preferably normal distribution and if it’s not normal make sure you make it normal distribution. Or even the other distributions are used as in when needed or if more suitable. And obviously the distribution is supposed to be normally distributed for a given class not he entire feature, if the N classes are there then all numerical feature values corresponding to those N classes should follow the distribution separately.

Obviously, there is another method of creating the categories by binning and KDE can be used and so on.

Assumes the independence of the features needed for the proof and that is why called as the naïve and gives equal weightage to all the features.

There are different types of the naïve bayes based on the distribution of the variables:

For numerical as well as categorical features together we do not have a specific naïve bayes in sklearn

Better use

from mixed\_naive\_bayes import MixedNB

other implementations in the SkLearn are

based on the distributions:

Bernoulli

## **Decision Tree:**

* No preprocessing needed like scaling, distribution skewness etc unlike other algo’s.
* white box model, result is very much interpretable.
* But it has huge tendency to overfit.
* Not stable nor continuous output prediction which makes it difficult to use for extrapolation.
* DTs are very unstable, even the slight change in the input data would make entirely different DT which cause high change in the output for very small change in the input.

**What do DT’s do?**

Let’s take an example, we’ve a data of personal showing their expenditure characteristics and personal demographics based on which we would like to predict whether the personal is Extravagant spender or careful spender. Data includes personal info like Name, Age, Gender, Address, Urbanicity, spending chars etc. Now let’s suppose we decides to separate the target variable based on single variable, like urbanicity in above case, we’ll have Extravagant spender in urban and careful spender in rural, so even based on this single column we can very much predict the outcome if all the Extravagant gets into rural split and the careful ones into rural split. Obviously, we won’t have pure nodes after this split and then on next splits we’ll try to split this individual split on the next best features and ultimately reach the pure nodes. i.e. DT’s.

**Intuition:**

Viz this data and try to see which feature is split the data so that we have bast segregation of the target variable. i.e. each split is pure or as close to pure as it can be. And the vanilla/ traditional DTs tend to split till each node/split is pure.

The goal in the DTs is to find all the possible branches, pathways to reach pure node with homogenous datapoints. This branches which reach pure node are created by splitting the data into more similar section (tends towards homogenous). These branches are created based on the feature values, and while prediction based on the feature values branch is traversed and corresponding node helps predict the output.

Geometric Intuition: Geometrically Decision trees splits the region with plane orthogonal to the dimension/axis with each branch creation and regions are formed in a such way that homogenous points are clubbed together, and more branches you create/ deeper you go in DTs more and more subregions are created where each deeper region is more and more homogenous, at the deepest level of DT with max branches you have regions which are purely homogenous which are pure nodes itself. This recursive partition makes rectangular regions in planer region whereas hyperrectangular regions in multi-dimensional decision space.

Actual Algorithms: so, we basically want to create this branches / region based on feature values which are all the possible unique ways to reach a pure node.

Classification: The basic underline rule is you need to split the data into more homogenous subsets. The way to do it is, you start with feature with a feature value that splits the data in best possible homogenous subsets i.e. you split based on the feature and feature value which lets you make the splits with least entropy / variance in the subsets. i.e. more homogenous subsets.

How do you judge the entropy, variability, randomness in the subsets based on the feature and feature value?

In classification problem there are mainly tow ways measure the entropy,

1. **Gini impurity:**

Calculates the probability of misclassification of the randomly selected element if it were randomly labelled based on the class distribution in a subset. It indicates the degree of split, or the quality of the subsets created. It indicates the mixture in the subset, all the classes are same then the Gini impurity would be lowest. Value of the Gini impurity varies between the 0 and 0.5 for the binary classification, where 0.5 being the worst value with maximum mixture and creating this subset is not helpful irrespective root Gini impurity score.

On fundamental level it tries to answer the similar information like entropy which is the degree of randomness and mixture in subset. Though there is slight change in which both do it.

The range of the Gini impurity value is based on the number of classes and given by:

1-(1/n) i.e. for 2 🡪 1-0.5 = 0.5

* (1-1/3) = 0.66
* And 0.75 for 4
* has option to select entropy.

Gini impurity (Ho) = Gini impurity indicates the probability of misclassification of randomly selected element if the elements are labelled randomly (randomly chosen from pool/dataset i.e. proportional probability).

It indicates the similar thing as indicated by the entropy, which is randomness, mixture, inability to predict correctly.

Gini impurity = 1 **-**

Gini impurity = 1 - (p1^2 + p2^2)

Formula breakdown:

First let’s understand what is randomly labelled. Randomly labelled means randomly a single element is picked, i.e. proportional probability of each element. i.e. if there are 100 balls of Red, Pink, Yellow with 60, 30 and 10 samples respectively. Then on labelled randomly or one is picked randomly then what is the probability, for red = 0.6, pink = 0.3 and for yellow = 0.1.

Now the correctly classifying red as red, pink as pink and yellow as yellow with this proportional char is given as p(r) \* p(r) + p(p) \* p(p) + p(y) \* p(y)

i.e. 0.6 \* 0.6 + 0.3\*0.3 + 0.1\*0.1

and now the probability of misclassifying the elements is 1- this above value.

This is how we get the metric of randomness, inability to predict, entropy, mixture is obtained.

Used for CART

1. **Entropy / information Gain:**

LOG: what are logs? The basic intuition of the log is that they describe what power of the base is the required value. i.e. LOG (100) i.e. logarithmic value of 100 with base 10 is 2, because it tells what power of base it needed to get the 100, which is 2, 10^2. Same for all the logs, for ln it finds what power of e i.e. 3.71 is needed to get the required specified value. Log for 1 with any base is always 0 cause of the same and 1 is logarithmic of value same as base.

Make sure to watch intuitive content of log.

Entropy: Randomness / mixture/disorder/information/surprise in the system is generally referred as entropy, system could be any statistical, physical (thermodynamics). Entropy indicates how uncertain the system is or how unpredictable it is. Most important definition: In information theory, i.e. in our context entropy **indicates the average amount of information or uncertainty in random variables**. And basically, tells how unpredictable the outcome of the process is. This entropy is given by Shannon entropy formula which is:

In the above formula the p(x) indicates the probability of the outcome whereas the log2(p(x)) measures the information or surprise element of the outcome, lower the prob higher will be the surprise element of that outcome.

Entropy for 2 outcomes, i.e. tossing a coin,

= 0.5(log2(0.5) + 0.5(log2(0.5) = -1 = 1

This is max entropy of the system in binary output problem, same for the 3 output it is log2(3) = 1.58 and dice it would be 2.58, log2(6) and 4 is 2.

Why is entropy important? Because it determines the amount of information needed to describe the data.

How come the entropy is average of the information or uncertainty or surprise? Information term is weighted by the probability, means only the respective required proportion of the information is considered.

**Information is surprise is uncertainty. Which is always the higher when we have equal likelihood and no bias. i.e. system is most random for no bias and equal likelihood. And information, uncertainty, surprise must be lower as low as possible.**

Information tells how much the uncertainty of this event is, calculated with the help of the probability of the event and weighted by probability itself. Smaller the probability, rarer the event higher is the information,

Information gain is reduction is entropy or average information after the split about the dataset.

Used in ID3 and C.45

1. **Mean squared errors (variance): For regression.**

There are few DT algorithms based on how they split the data and creates branches:

* CART: is more advanced version which uses Gini impurity and do only two splits at each node, has inherent ability to work with both categorical as well as numerical values. In sklearn the implementation of the DT is though CART only. Though CART has option to select entropy as optimization metric.
* ID3 used entropy which as metric to decide best split where the number of splits for categorical are as many as the number of categories in a feature.

Let’s start with CART. (Classification and regression trees):

Let's go through the detailed calculation again, considering the binary splits for Feature C as described. We'll use the same dataset:

Check the tree viz. in notebook for better understanding.

| **ID** | **Feature A** | **Feature B** | **Feature C** | **Class** |
| --- | --- | --- | --- | --- |
| 1 | Cat1 | B1 | C1 | Yes |
| 2 | Cat1 | B1 | C2 | No |
| 3 | Cat2 | B2 | C1 | Yes |
| 4 | Cat1 | B2 | C1 | No |
| 5 | Cat2 | B1 | C3 | Yes |
| 6 | Cat2 | B2 | C2 | No |
| 7 | Cat1 | B1 | C1 | Yes |
| 8 | Cat2 | B2 | C3 | Yes |

Step 1: Root Node (Level 0)

Calculate Gini Impurity for the whole dataset:

Total instances: 8

Classes: Yes (5), No (3)

Gini = 1 - (5/8)^2 - (3/8)^2 = 1 - 0.3906 - 0.1406 = 0.46875

Evaluate each feature for splitting:

Feature A:

Cat1: Yes (2), No (2) -> Gini = 1 - (2/4)^2 - (2/4)^2 = 0.5

Cat2: Yes (3), No (1) -> Gini = 1 - (3/4)^2 - (1/4)^2 = 0.375

Weighted Gini = (4/8)\*0.5 + (4/8)\*0.375 = 0.4375

Feature B:

B1: Yes (3), No (1) -> Gini = 0.375

B2: Yes (2), No (2) -> Gini = 0.5

Weighted Gini = (4/8)\*0.375 + (4/8)\*0.5 = 0.4375

Feature C:

C1 vs. not C1:

C1: Yes (3), No (1) -> Gini = 0.375

Not C1: Yes (2), No (2) -> Gini = 0.5

Weighted Gini = (4/8)\*0.375 + (4/8)\*0.5 = 0.437

C2 vs. not C2:

C2: Yes (0), No (2) -> Gini = 0

Not C2: Yes (5), No (1) -> Gini = 0.277

Weighted Gini = (2/8)\*0 + (6/8)\*0.2778 = 0.208

C3 vs. not C3:

C3: Yes (2), No (0) -> Gini = 0

Not C3: Yes (3), No (3) -> Gini = 0.5

Weighted Gini = (2/8)\*0 + (6/8)\*0.5 = 0.375

Best Split: Feature C with the split 'C2 vs. not C2' gives the lowest weighted Gini impurity (0.2083).

Resulting Tree (Level 1):

Root splits on Feature C:

C2: {2, 6} (Yes: 0, No: 2)

Not C2: {1, 3, 4, 5, 7, 8} (Yes: 5, No: 1)

Step 2: Split on Not C2 (Level 2)

Not C2 Group:

Gini = 0.2778 (from above)

Split on Feature A:

Cat1: Yes (2), No (1) -> Gini = 0.4444

Cat2: Yes (3), No (0) -> Gini = 0

Weighted Gini = (3/6)\*0.4444 + (3/6)\*0 = 0.2222

Split on Feature B:

B1: Yes (3), No (0) -> Gini = 0

B2: Yes (2), No (1) -> Gini = 0.4444

Weighted Gini = (3/6)\*0 + (3/6)\*0.4444 = 0.2222

Best Split for Not C2: Both Feature A and B give the same result; choose Feature A for simplicity.

Resulting Tree (Level 2):

C2: {2, 6} (No)

Not C2 -> Cat1: {1, 4, 7} (Yes: 2, No: 1)

Not C2 -> Cat2: {3, 5, 8} (Yes: 3, No: 0)

Step 3: Split on Cat1 under Not C2 (Level 3

Cat1 Group under Not C2:

Gini = 0.4444

Split on Feature B:

B1: Yes (2), No (0) -> Gini = 0

B2: Yes (0), No (1) -> Gini =

Weighted Gini = (2/3)\*0 + (1/3)\*0 = 0

Best Split for Cat1 under Not C2: Feature B.

Resulting Tree (Level 3):

C2: {2, 6} (No)

Not C2 -> Cat1 -> B1: {1, 7} (Yes)

Not C2 -> Cat1 -> B2: {4} (No)

Not C2 -> Cat2: {3, 5, 8} (Yes)

Step 4: No further splits needed for this example since we've reached leaves where no further beneficial split can be made given our small dataset.

Summary:

Here, we've gone through four levels of splitting, selecting the best binary split at each step based on Gini impurity.

The tree stops growing when no more splits would decrease impurity or when all instances in a node belong to the same class.

For numerical feature a middle feature is ordered in ascending order and a split is made based on every middle value between the two subsequent values.

Complete this CART’s rest info after viz in splits in notebook.

Feature importances.

**In actual implementation of the CART in normal decision tree, random forest, gradient boosting and in standard part of the XGBOOST there is nothing as categorical handling like shown above, This vanilla CART knows only the numerical understanding of the features, as encoded in preprocessing only and it process everything in similar manner of how it does the split on numerical column, so even if you label encode or category encode these features it treats then the same.**

**Only in LIGHTGBM and in XGBOOST’s with its experimental parameter we can have categorical features treated as categorical with not equal to operator. This is one of the biggest advantage of the LIGHTGBM.**

**Regression:**

for regression, the metric to judge the impurity is variance (MSE in this case, as the prediction for entire split/leaf is same which is mean). Even MAE can be used to judge the split.

**Feature importances in Decision Trees:**

Compute the impurity reduction like you calculate during forming decision trees. Do the normalized sum of this delta impurities for each node where the split is made by the feature for which you are calculating the importance. Total sum of this values will give you the feature importance of that value. Then this feature importances for each feature are normalized with respect to the total sum of the features importances, so that we have feature importances of all features which sums to 1.

For normalization in delta impurities for each node, do this with respect to all the datapoints (i.e. root node and not the immediate previous local root).

i.e. why first split after root node will have weight of 1. Cause Nnode and Ntotal is same. Which leads to a point that this feature will have the highest feature importance most of the time. But not always, mostly in complex datasets it’s not the case. If the first split has very high delta impurity and when the trees are shallow i.e. other feature do not get that much chance to get split on.

But in deeper trees where lot of feature get’s split on there might the case that weighted delta impurity in total might be highest for some other feature which is not used for fist split.

Formula can be referred in the notes and which is for each node, you sum it for the all the node splits for each feature.

Hyperparameters of the Decision Trees:

1. **Depth:**

Decide the maximum depth of the search or decides the max path of leaf node from the root node. Limits how deep trees can grow. Gives you the ability to make either simpler or complex tress depending on the depth. Which eventually controls the underfitting or overfitting. Obviously shallow trees will be simpler with underfitting whereas the deeper trees are more complex and captures more complex patterns and hence are leads to overfitting.

* If the one of the branches reaches the leaf node prior this does not mean we stop, branches are kept on forming where they can be formed till the specified depth.
* Default value is None i.e. max depth is till the leaf nodes are reached. For simpler trees i.e. underfitting trees value of depth is 3-10 and for deeper trees the same value is between 10-30, but obviously this entirely depends on the dataset.
* Tuning can be done with cross validation with hyperparameter tuning for different possible values and the graph between the accuracy (any metric) and the possible van be seen to how the metric is getting affected.

1. **Min samples split:**

* If the no. of samples in any of the nodes are greater than certain threshold limit then, no split is done in this case.
* Intuition is pretty much the same as above, about how it effects, higher the value of the threshold lower will be the depth.
* Absolute min value which can be set is 2, which is the default in the sklearn’s CART implementation.
* Common thumb rules, if the dataset is larger the min samples in a node allowed are bigger compared to the smaller dataset for obvious reasons.
* Actual range for smaller dataset is 2-5 or at max 5-10 for slightly medium dataset, whereas for larger dataset it is 10-20 and 20-50.
* This min samples can also be specified in terms of the percentages of the dataset size, for smaller datasets it is 0.5%, then you go on towards 1% and then to 5% for medium datasets. And for larger dataset value up to 10% can be tried. There is no special parameter defined, you do this calculation prior and then specified the inter value.

**[2, 10, 50, 100, 500] or [0.01, 0.05, 0.1])**

1. **Min samples leaf:**

* Like the min sample split, jus the difference is you put the criterion on the leaves which are getting after the split rather than putting criterion on nodes itself. So, potential leaves sizes are constrained rather than simply constraining the node itself.
* The min value here is 1 which is also the default value in CART implementation, the same value was 2 in case of the min sample split.
* Values are 1-5 for small dataset, 5-10 for medium and 10-20 and 20-50 for larger,
* In terms of percentages values are 0.5%-1% for smaller dataset, 1% - 2% for medium and 2%-5%, 2%-10% for larger dataset.
* Difference between the min samples split and min samples leaves, first one i.e. min sample split tells that do I have min samples in a node to even consider split whereas min samples leaves says will be potential new leaves will have enough samples, node in this case should have at leas samples equal to 2 \* min sample leaves.

1. **Max number of features:**

Max number of features which are used for splitting each split is restricted to this number and these features are randomly selected, if the unique features remained or max features specified are higher than available features for split then in that case all the available features are used. Why is it helpful, it adds that level of randomness which makes the model more normalized and reducing the overfitting.

Max features are specified as [None, "sqrt", "log2", 0.1, 0.5, 0.8].

1. **Max leaf nodes** Counts the max number of leaf nodes in the tree and based on this criterion decides the growth/pruning of the tree. But it only concerns the leaf nodes and not the internal nodes. Values possibly set are None, 5-10,20-50 depending on the data.
2. **Min impurity reduction:** If the impurity reduction in next split is not above the specific threshold, then in that case the split is not done.

* Possible values are: [0.0, 0.001, 0.01, 0.05, 0.1]. Where 0.0 mean no stopping criterion and the 0.001 to 0.01 is for moderate criterion whereas the 0.05 to 0.1 pretty strong pruning with very high normalization.

1. **Criterion:** For classification, Entropy/Gini Impurity. Gini is bit fast as no log whereas entropy favours the balanced dataset. Though Hadley any changes for classification whereas for the regression, we have variance mainly but MAE and poissions can be used.
2. CCP alpha: How does CCP alpha works? How does it prunes the trees, How does it do the post pruning?

**Make sure you understand the difference between this different hyperparameters.**

## **Ensemble Learning:**

Ensemble techniques use principle of wisdom of crowd to correctly identify patterns from the crowd. It involves predicting the same value by multiple models i.e. by crowd.

There are multiple ways in which you create this crowd or multiple models. The fundamental assumption while creating this model is every model should be totally independent from each other.

**Why do ensemble work?**

Ensemble techniques work because they exploit the statistical, computational, and representational principles to improve predictive performance.

1. Error reduction through Aggregation and Diversity.

Let’s answer it later.

Based on how it creates different models, there are four types of ensemble techniques:

Different models are created such that they do not have correlation between them.

### Voting:

Most intuitive of the ensemble techniques where different models are trained on same data. That is how it creates the decorrelated models. There is also option of sampling the data, just to create additional layer of the independency between the models.

Very basic and democratic way to include the different voters to predict the output by aggregation.

**Basic concepts:**

Usually the base learners (different models, or same with different parameters) are trained on the same data usually. And these outputs are combined by aggregation or majority count.

Types of the Voting classifiers:

**Hard Voting:** Each model makes the single definitive prediction, and output is the major count of one of this prediction.

**Soft Voting (relevant to Classification):** Each model returns the probabilities, and these probabilities are aggregated for prediction, this is the main advantages of the soft margin as hard margin is not able to capture the essence of higher likelihood of its prediction. 51% chance of prediction is same as 99%. But the disadvantage is each model should be able to return the probability.

Even the weights could be assigned to each of the model before aggregation it has higher impact of the specific base learner, though it is not stacking cause stacking has one more layer of the learning.

Even sampling is possible in some cases.

Assumptions and requirements:

**Diversity:** All the voters should be highly diverse; cause voting works best when each model makes different kind of mistake on data.

**Independence:** The errors made by the models should be highly uncorrelated.

**Competency:** Each model should be at least slightly better than random guessing, then only combining lot of them.

**Why does it work?**

**Errors are cancelled out,** cause all the models do not make same mistake,

**Strength amplifies,** Different models capture different patterns in the data.

**Implementation, Voting Classifier ().**

### Stacking:

Multiple models are combined such a way that their output is given to another model which weights the output of all the models and then those weighted outputs are combined/aggregated.

Stacking is an advanced ensemble learning method where the models (base learners) are combined by training the meta model to learn how to merge this different model together.

The only difference between the voting and stacking is how you combine this base learner’s prediction.

**How is the meta model developed or works?**

* Train your base model.
* Generate the OOF prediction for each meta model with cross validation for each base model.
* Now this OOF predictions of each base learners becomes the training data, and the target is the original target feature.
* So, what happens is for each base learner you get as prediction for each, and every training data points through CV OOF. And now you as many predictions for each training data point as number of base learners used. And the problem to solve now is while aggregative this prediction which of these prediction to weight what. So basically, you want to find the relation/impact between the prediction (level 1 prediction) by each model and actual target.
* So, for the same a new model is trained called as meta model, which finds how relate this prediction of each base learner and the actual target. Which one to what weight for what kind of original data.
* Suppose the meta model is linear then meta model learns, how to weigh base learners or how to combine them unlike voting where you directly aggregate, in linear model specifically finding the weight of aggregation means finding the features importances of the features in meta model training. For non-liner models like random forest or GD, non-linear patterns are found out like, for certain kind of data certain base learner works best then it weighs that base learner higher importances and makes prediction likewise, then more certain ranges where one model could be doing fine but the other may not be doing any good.
* So, the actual process is you passed a datapoint though each base learner then passed those prediction of the same point though meta model and finally meta models’ prediction is the real model.

How does training data looks like for meta model:

* Let’s suppose we have 3 base learners and 100 datapoints, for 3 base learners with hard predictions, data looks like this for 20% test set.

80 X 3 – input data

80 X 1 – Target variable, original training set.

* For soft margin with multiple probabilities, its columns increase based on which probabilities you are increasing.

Advantages:

* Captures complex relations between the base learners, i.e. non-linear weights unlike voting.
* Outperforms voting usually.
* Adaptability to various scenarios.

Disadvantages:

* Risk of overfitting.
* Complexity is high.

Practical tips:

* Base models: use wide variety, or diverse models.
* Use simple meta model to avoid overfitting.
* StackingClassifer () is the implementation in sklearn.

### Bagging. (Bootstrap + Aggregation)

Ensemble learning techniques introduced by the Leo Breiman in 1996. The core idea is to reduce the variance of the model’s prediction by training the same model on multiple subsets of the data and then aggregating it.

How does it work?

Bootstrap sampling: sample the data with replacement, one sample for each model.

Train base learner: train single kind of base learner on this model.

Aggregate predictions: aggregate the prediction of each base learner.

**Intuition:**

Very high variance models, like DT’s are used as base learner which overfit on the sample data, mining all the intricate patterns in those samples, but the prediction/trees for each sample could be totally different as this overfit model are very sensitive to variance. Means, each base learner with its sample data digs deep and finds new patterns but this DTs are different for each sample.

Hence on averaging this common pattern gets highlighted whereas the errors get overshadowed.

Viz a one chunk of the data creates one decision, and another chunk creates another one. Like this each tree zigzag the noise in the data.

**Why does bagging works:**

* High variance models.
* Averaging reduces the variance.
* Bias stays roughly the same.

**Key characteristics of the bagging:**

* Parallelizable
* No overfitting Penalty
* Best with unstable learners.

**Advantages:**

* Reduces variance.
* Robust, improves stability.

**Disadvantages:**

* Does not reduce bias.
* Computationally heavy
* Single DT has better interpretability.
* Correlation between the points, should not be high correlation between the samples, could be the issue in smaller datasets.

**Why is bias called bias?**

Bias indicates the bias nature of the model to predict values towards its ideal simplistic nature, like linear regression is bias towards linear relationships irrespective of the real nature of the data. Naïve bayes is bias towards the feature independency. This is the start of the origin of the concept, bias indicates how good model can fit on intricate patterns with no restrictive assumption. High bias means high inclination of the model towards the restrictive assumption, very simplistic model like linear regression. High bias means models simple model which makes the mistakes on the training data itself. Whereas the low bias means model does not incline towards the specific restrictive assumption.

**Bootstrapping:**

Sampling with replacement is done in bagging. Usually each sample (goes to each estimator) is sample with replacement and usually the size of original dataset only. With replacement creates bit more of the randomness though it has repeated rows.

With replacement and without replacement?

With replacement means on drawing a sample you replace the original pool with same sample, so the next picking would have same probability of picking of each datapoint as for first drawing of the sample.

Without replacement after each draw, you do not replace the removed datapoint with anything. So, for each subsequent draw the probabilities changes.

For N datapoints if you draw a sample of N datapoint with replacement then out of the 100 unique datapoints there would 63.4 % unique datapoints would be there and remaining would be repeated.

Max\_samples in the sklearn decides the sample size, default value is ‘None’ means equal to pool size means equal to the original dataset.

Bootstrap is another parameter which decides the sampling with and without replacement. Default is True which means sampling with replacement. And make false, you change the max\_samples as well, this is the general way to use it else it will just give same data to all the trees.

**Types of bootstrapping:**

1. **Bootstrap bagging:**

Sampling with replacement, very typical case bagging with sample size same as initial data.

1. **Subsampling Bagging:**

Without replacement bagging with sample size lower than initial data. i.e. if the dataset has 100 datapoints then out of this 100 datapoints, each sample will have 80 datapoints only but without replacements.

1. **Pasting (disjoint sampling):** like above subsampling bagging just the sample size is divided such that there will be no overlap of the datapoints across all the samples. Totally unique and disjoint samples.
2. **Random subspace:** sampling on the columns, always without replacement.
3. **Random patches**: combining bootstrapping (sampling with replacement) with random subspace (without replacements)

**What is the difference between the bagging with DT and Random Forest?**

For subset sampling random forest uses, normal bootstrap bagging (Row sampling with replacement), but it does column sampling on each node for split. i.e. column sampling happens at the tree formation stage. Th

No. of columns selected, for classification it is square root of no. of columns whereas for regression it is no. of columns / 3.

This number is higher for regression cause the regression can handle higher correlation between the models, as the final output gets averaged. But the same is not possible to an extent as it is possible on regression for classification cause outputs do not get averaged, here value count wins.

Technically higher number of features cause lower decorrelation, so we desire lower no. of features, yet we want as much info as possible with lower noise means max features, so we want max decorrelation with min features.

In classification the outputs are aggregated by majority count means the extra decorrelation caused by having more features for extra info and lower noise cannot be adjusted by aggregating the outputs, which is very much possible by the aggregating the regression outputs, that’s why we allow higher features (M/3) cause regression can handle bit for the decorrelation, because of its ability to handle the decorrelation in the output aggregation.

**OOB score:** sampling with replacement leaves about 33.2 points out of the sample, so the prediction is done on this non-used datapoints, to get the evaluation idea.

How it is done, take each of the datapoint one by one from the original sample and then check in which of the sample it was not selected, then use those respective tresses to predict the output of this datapoint and aggregated those outputs.

It helps you not use validation set. It is like test set, i.e. in terms of data leakage and fundamental calculation method, just the trees used to predict the point are not all, only the subset of the trees is used, which obviously may not be the best estimate of the datapoint.

**Why it is useful?**

* No need of validation set.
* Hyperparameter tuning.
* Efficiency (no extra calculation)

When you do the oob\_score = True then the OOB score during the training is shown.

Hyperparameters of the random forest

Extra-Tree Classifier: splits do not use impurity metric to judge which metric is split is better, random splits for better decorrelation.

### Boosting:

Why boosting is most of the time better than bagging:

* Bagging relies heavily on the base learner ability to capture the patterns, i.e. it relies on base learners being extremely strong, or their inherent ability to be extremely strong. If the base learners are not strong then bagging averages out this weak prediction for variance reduction (Generalization).
* Boosting focus on the extreme/outlier, different/difficult to predict points whereas as the bagging tries to average them out in aggregation stage. Whereas boosting explicitly focuses on these points either with weights or in residual fitting parts itself (next models training). Boosting focuses on the difficult problems(points/patterns) specifically whereas the bagging does not.
* Boosting can optimize any complex differential function, same cannot done in bagging.
* Though bagging is better when there is a risk of overfitting whereas the boosting overfits very normally as it tries to add the sequential models to reduce the any errors, where bagging does reduce the bias by aggregation significantly.

Baggins is faster as it can be parallelly performed whereas the boosting is sequential adding of the models and hence slow.

#### **How boosting works:**

It works completely in opposite way than how does bagging works, in boosting you start with very simple model with very high bias and low variance model, and then tries to improve the bias by adding the weighted sequential models, where each new added model trains on the residuals of the previous model and tries to reduce this residual as much as it can.

These models are weighted based on the relative performance in reducing the residuals (relative error reduction with respect to the whatever residual is left to it, cause the absolute residual reductions are always higher for earlier models.

Weights are lesser than 1 means model required to converge the solution are more than what required without the weights as with weights you only take/add certain proportion of the residual reduced.

#### **Gradient boosting algorithms:**

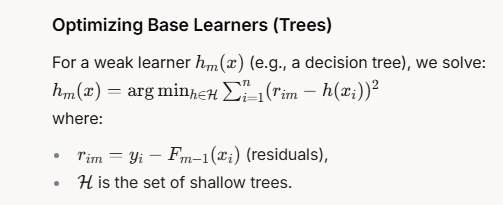
Now we know how boosting works? Let’s see how everything is implemented.

The real algorithm stepwise:

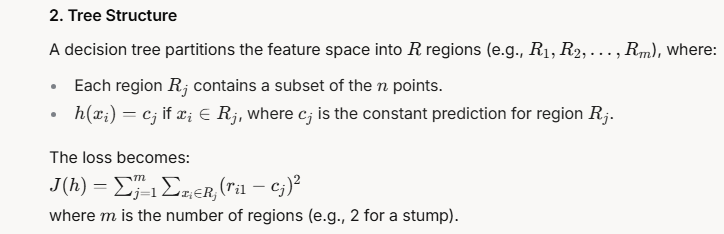
* + First initial model. The initial model is always the constant model (function) which tries to optimize the loss between the actual target values and the model prediction.
  + For the actual formula and calculation with MSE as a loss function see the notes, which involves the defining the loss function for entire data with respect to the initial function(model) and optimizing this loss function with respect to this initial function which is to be found (decided by solving this equation only) you get the initial function. How do you optimize it, by gradient descent i.e. derivative minima optimization. For the MSE as loss function, we get initial model is mean. We directly add the mean or initial model as it is not the improvement step in correction of the loss as it is the first general prediction.
  + Once the first model is calculated then we add the weak learners whose aim is to predict the residual (Ri) between the initial model and the target variables. Weak learners DTs takes the same Input X and target as the residual (Ri) and try to predict best residual (remaining part of the prediction) by finding the most optimized function/model/weak learner. But unfortunately, the constraint on the weak learners is that we mostly have DTs as weak learners and this multiparameter models are not differential, so we cannot directly use the gradient descent as we used in the case of the initial model.
  + There is as such requirement of only having trees as the weak learners, there are only two requirements as such, the weak learners should be weak, slightly better than random guessing and second criterion is that these learners should be calculated from the x (i.e. input) so that they can be input/data specific. But why do we select decision trees, due to specific advantages:
    - Natural fit to gradient approximation: Trees partitions the data into different regions and the prediction is done based off the region mean, this piece wise constant fit aligns with the approximation of the gradient direction simple interpretable way.
    - Non-linearity and flexibility: can efficiently capture non-linear pattern yet can be made simple(weak) without feature engineering
    - Weakness control: High and direct easy weakness control.
    - Scalability:
* Other options of the base learners:
  + Linear model
  + NN
  + Splines
  + KNN etc.

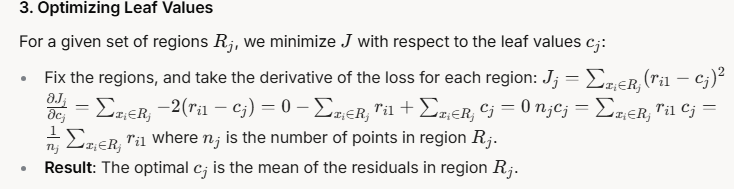
But we do select the DTs, let’s see how we find out the best DTs or most optimized DTs in terms of right direction and right magnitude to reduce the residual given by the previous step. Anyways that’s the job of the DT (base learner). If they were differentiable, we could have been able to use gradient descent as it was used for initial model, but here we do the optimization in a different manner, which is explained like this.

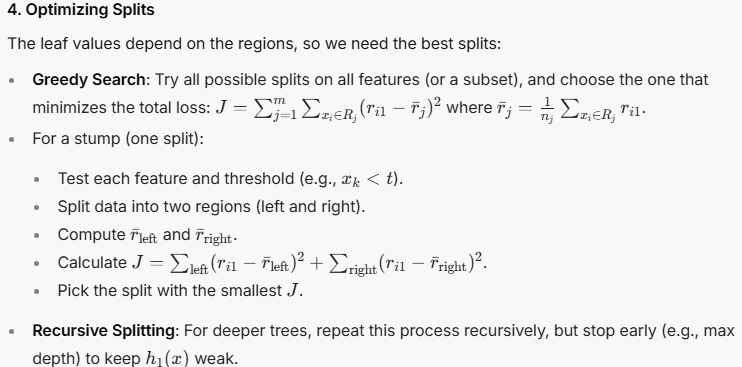
Optimization function with loss is given as this, where base learner is selected from the pool of base learner H, where the best learner which minimizes the loss (residual).



Finding the best “h” is a two-step process first finding way to aggregate the regions points and second the optimum number of regions or leaves.







This is how the best most optimum DT (base learner is selected) with mean as aggregation (regression) and best no. of split for regions are selected.

* So, what we do is we find the most optimum h1 and then take only the weighted proportion of it, which is analogous to taking step in the right direction but only the limited proportionate step?
* Weak learner is already a shallow tree and then we again add the weighted portion of it?

**Weakness Alone Isn’t Enough**:

* A shallow base learner is intentionally simple to avoid overfitting—it captures only major patterns in residual​. But even this simple correction might be too aggressive if added fully.
* Example: If base learner overcorrects a noisy pattern (due to its fit-on residual), adding it fully could harm generalization.
* **η Adds Fine Control**:
* Iterative **Refinement**:
* The learning rate η \eta η acts like a “dimmer switch” on h1(x) h\_1(x) h1​(x)’s contribution. Even though h1(x) h\_1(x) h1​(x) is weak, scaling it down ensures we don’t trust it too much in one go. Analogy: h1(x) h\_1(x) h1​(x) is a rough sketch of the correction direction; η \eta η decides how much of that sketch we paint onto the canvas at once. Gradient boosting builds a strong model by summing many weak learners: Shallow trees ensure each hm(x) h\_m(x) hm​(x) is simple; η \eta η ensures each step is small, allowing the ensemble to refine gradually.
  + Balancing Bias and Variance: Shallow model High bias (underfits residual​), low variance (stable).
  + Full step (η=1 \eta = 1 η=1): Risks increasing variance by relying too much on one weak fit.
  + Small η \eta η: Keeps variance low, reduces bias slowly over many iterations.

so, what we do is we find the most optimum h1 and then take only the weighted proportion of it, which is analogous to taking step in the right direction but only the limited proportionate step.

Trees are kept shallow in learning phase by constraining the overfitting parameters.

* **Numerical calculations in the notes.**

#### **For classification (Difference specific to classification):**

Gradient boosting algorithm for classification works just fine, like it works for regression except loss function it uses. In classification the loss function used is log loss error and then onwards the subsequent steps are similar, means the first constant model is developed with log loss error, and it comes out to be log of odds. Log of odds, log (probability of yes/ probability of no), this yes and no are prior probabilities, i.e. count of yes/total and count of no/total

Then for the base learner, optimized base learners are calculated, the method to get the optimized DT is same, just the loss function is different so, the way to calculate the log of odds for a node or rather way to aggregate in classification to find the log of odd of the node (cause DT gives log of odd, and not the probability. Log of odds is calculated as sum of residuals to the sum of variances (or sum of their uncertainties).

And this log of odds then changed to probabilities and then compared to get the residual,

The model in gradient boosting for classification does not directly predict the residual as the difference of probabilities. Instead, it predicts adjustments to the log-odds, which are then transformed into probability changes, though it takes difference of the probabilities as the input.

#### **Why gradient boosting is a gradient descent in a function space?**

Gradient boosting mirrors gradient descent by iteratively stepping in the direction that most reduces the loss, but it does so by adding functions rather than tweaking a finite set of parameters. The weak learners act as approximations to the steepest descent direction in this abstract function space, making the process both powerful and flexible for a wide range of problems.

### XGBOOST:

Curvature: slope is the rate of change of the function whereas the curvature is the rate of change of the slope itself. Tells if the slope is changing at the point or no and the magnitude of it.

High curvature (large (x)): The function bends sharply, so small steps are safer. Low curvature (small (x)): The function is flatter, allowing larger steps. Cause for higher curvatures the values change rapidly, and could be rapidly overshoot, so for higher curvature the steps are little.

This curvature is represented by hessian matrix, and the xgboosts optimization is based off this second order function optimization unlike the gradient boosting.

#### **Core idea of XGBOOST:**

* **Speed and efficiency:** use parallelization, tree pruning and hardware optimization.
* **Regularization:** uses L1 and L2 regularization penalties to prevent overfitting.
* **Flexibility:** supports custom functions and handles missing data natively.
* **Scaling:** Can be used on large dataset.

#### **Mathematical foundation:**

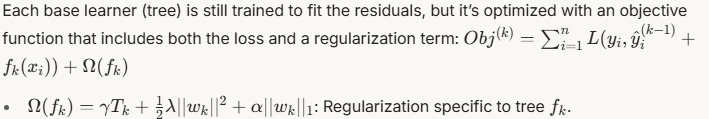
1. **Regularization in each base learner:**

The main mathematical difference in gradient boosting and XGboost is that XGBOOST add regularization parameter along with loss function and the combine optimization is performed.

Optimization function for the base learner to add is combination of the loss function + Regularization parameter. Means the optimization for the base learner (i.e. finding most optimum base learner) is based of loss function + regularization parameter, means regularized base learner is added.

So, that’s the main difference between the GD and XGB, that regularized base learners are added in the XBG whereas in GB non regularized base learners are added.

Objective function given below with regularization function



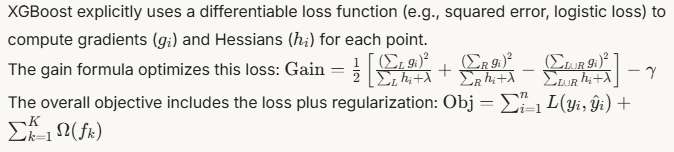
1. **Second- Order Gradient (Hessian) in optimization:** It also uses with hessian derivatives (curvature) for finding the optimum solution with Taylor series expansion (for faster approximation of the solution).
2. **Customizable loss function:**

Unlike the traditional GD, XGBOOST can handle any customize loss function which is differentiable of first and second order. This flexibility is due to the second order Tylor approximation method.

1. **Weighted quantile sketch for split finding:**

Unlike traditional DT which finds the best split after calculating all the possible splits of the numerical values of all the features, XGBOOST uses the smart method of dividing the features values into a quantile buckets, and these buckets are formed on the cumulative sum of the hessian values. The number of the buckets are dependent on the sketch eps, where the number of buckets is given as 1/sketch\_eps. Sketch\_eps determine the allowable absolute error in terms of the cumulative values in quantile approximation. Then the splits are done only for these buckets and the best one with the max gain is selected. The hessian is calculated for each value of each feature by hessian formula of Pi(1-Pi) where Pi is by sigmoid (1/1+e\*i). numerical process involves calculating he total hessian then dividing the total hessian by number of buckets to get the bucket limits, then grouped the points falling into each bucket and find the gain for each bucket. Finding gain here is nothing but finding the reduction in loss. Which does not involve the proportionate taking difference between the root entropy or loss and sum of left + right after split, the weight proportionate is already there in the gain of right and in hessian values itself. The formula for the gain is bit different with ½ at the front.

The **main idea** behind the concept is XGBoost splits by grouping points into buckets based on cumulative Hessian weights, which highlight regions of significant curvature. It then selects the split between these groups that maximizes gain, ensuring the greatest reduction in the loss function.



1. **Sparsity-aware split finding:**

Ability to handle nan values and sparse in sparse matrix, so what happens is that nan values are treated such a that these values are put once to the left and then to the right whichever gets a better score is considered. In normal dense matrix an only nan values are considered for the above approach unless specifically specified about the zeros (converted to nan earlier only). For the sparse matrix the sparse as well as the nan values are treated similar and the score for both left and right with sending all sparse values (which includes sparse zeros as well as nan) is calculated.

1. **Tree pruning with gain threshold**
2. **Customizable Tree construction:**
3. **Hardware differences: use of GPU, Multi- core CPU’s, catch aware access, out of core learning,**
4. **Software differences: Built in Cross-validation, Early stopping.**

**Greedy search:** local optimum at each step and that is the global optimum is achieved no backtracking to explore all the combinations. NLP or LLM’s in general.

Brute force (exhaustive search): finds the global optimum with the trying all the combinations. Like hyperparameter with grid search CV

### LIGHTGBM

Five differences

* **Leaf split and not the stage level split.**
* **Histogram based learning.**
* **Categorical data handling based of importance to the accuracy.**
* **EFB (feature bundling).**
* **GOSS (Gradient base one side sampling)**

#### **Leaf base split:**

Unlike the normal DT split which is the case in XGBOOST, in LGB the split is not level base but rather leaf base. In leaf base split not each node of the level is split rather a node with which reduces the most entropy/variance is use for split, likewise all the leaves till the point are used irrespective of the level to judge which single leave reduces the impurity and this single leave is split.

Very unbalanced, non-symmetric tree compared to balance symmetric tree made by the level base split.

* E.g. of Level base split

Root

├── Left (Level 1)

│ ├── Left (Level 2)

│ └── Right (Level 2)

└── Right (Level 1)

├── Left (Level 2)

└── Right (Level 2)

Leaf wise split:

**Root**

**├── Left**

**│ ├── Left-Left**

**│ └── Left-Right**

**└── Right**

**Now for next split there are three leave are available left-left, left-right and right chose one to split.**

**Pros:** The main advantage is, it converges faster compared to the level base split, though there is not inherent advantage in term of split performance.

Though it can be overfit on the noise, so there is overfitting chance.

#### **Histogram base learning:**

The numerical variables are used divided into histograms, and the boundaries of those histograms are used for spit. i.e. only check the splits for the histogram boundaries.

#### **Categorical data handling:**

categorical data in the LIGHTGBM is handled without any encoding, categorical data splits are made just like the splits of simple explanation of the for categorical features in CART explanation. i.e. treat each category vs other with not equal to operator for all the possible combinations (not all technically) and then check the gain for each split. This is not what happens in the usual vanilla CART, which is used in the DT, ensemble like RF, GB, XGB etc. Each category is use for histogram like categorical feature treatment just the difference is it creates no bins rather uses each category as a bin and then computes the gradient and hessian for each category or bar calculates the gain based of these values. (just the difference from numerical is it uses categories for bins rather than step values of continuous variable). There is something called as **max\_cat\_to\_onehot** which decides whether the LGBs special technique is use for categorical feature handling or one hot encoding is to be used. This above parameter decides the max categories up to which the OHE is used, above this number of categories special handling technique of the LGB is used which tries all the possible combinations of the feature for split. For lower number of categories like below 4 (which is the default value of the parameter) OHE is used because it works better and is very simple and intuitive, higher number of OHE cause complexities and are far more computationally and memory wise difficult which is not the case for the lower number of the OHE.

Though again even for the higher cardinality it’s not like all the combinations are tried, cause for n=32 i.e. 32 categories in a feature (2\*31) -1 which is 2.1 billion combinations of the splits, not possible to do so there are few heuristic techniques it used for split. A parameter **max\_cat\_threshold** is use which decides the number of splits to decide int this case, default value is 32, which means only the 32 splits are made from the millions and billions of possible combinations based of the n. One more parameter here is after which value of N, does LGM starts to reduce the no. of splits out of all the possible ones, when the no. of possible split combinations gets beyond the 32 like in case of the 6 (63 only the 32 of these 63 will be check, obviously the best 63 would be selected by heuristic methods. For n=5, the no. of possible combinations are 31 so all these would be checked.

how do you select this best 32 split.

* One approach is one vs rest
* Top1 vs rest, (Top1, Top2 vs rest), (Top1, Top2, Top3 vs rest) this approach can be used.

#### **Exclusive feature bundling:**

The feature is bundled together which are mutually exclusive with their values. Like the mutually exclusive OHE values of the two independent features. Suppose we have feature called as the colours with 3 colour categories (Red, Blue, Green) and this feature is encoded with OHE then in that case we have 3 features (without dropping first) which perfectly mutually exclusive and these features can be bundles together. So, you process only one feature onwards in the data which is obviously computationally and memory wise efficient.

How bundling happens: For each datapoint the there will be only non-zero value out of these 3 features. That value is the value of these new feature for that row, if you just put the values as it is, then in that case the entire feature as single value 1, so that is why offset is used for each feature before putting into the bundled feature, or the index of column can also be put into there which keep track of the feature form this value arise which helps to track back to the original value with corresponding row and feature and that’s the ultimate goal, cause ultimately there values are tracked to the original feature and then then based on the corresponding target column values gain values are calculated. Offsetting is done such away,

Suppose features looks like

Red [**1**,0,0]

Blue [0,**1**,0]

Green [0,0,**1**]

Then offset for the blue would be 10 and for the Green would be 20 then after offsetting features looks like,

Red [**1**,0,0]

Blue [0,**11,**0]

Green [0,0,**21**]

The offset calculation is as such that it covers the entire ranges of the individual features.

And so does histogram bin calculation is based on this offset values so that single feature points get into one bin.

In case of indexing just the index of the column is mentioned.

How are this BF gets processed onward, there is only one histogram made for the bundled feature, based of this histogram and corresponding aggregates of the target, gain is calculated just like the normal XGBOOST, and the splitting is done. The main advantage of using the EFB is, it creates a single histogram which initially would have required 3. So EFB makes LIGHTGBM computationally efficient on the expense of the little accuracy if the conflict happens i.e. features are not totally mutually exclusive.

How does it handle the conflict, there is a parameter which decides the degree of the conflict, which is allowable, how rate of conflict is measured? it is nothing but the proportionate of the non-mutually exclusive rows to the total rows. Which feature values gets preference? the exact algorithm is not know but most likely the feature values with higher offset get’s preference, then the question arise how you decide the order of the features, ordering could be based off the

There are few nuances to cover about it,

First the feature preprocessing it does to identify the potential feature for bundling, it checks the sparsity ratios of the feature then then runs the patterns to check the mutually exclusive nature of the features.

#### **GOSS (Gradient base one side sampling):**

**It is** a technique in LGB used to improve the efficiency of the training especially on the large datasets. It makes the training on large dataset in terms of the computational and memory aspect for more efficient on small compensation of the accuracy.

**Core Idea:**  The core ideas are you select most of the points with high gradient but only the certain proportionate of the remaining low gradient with proportionate weightage is selected, the training is done on this small dataset which makes the training more efficient with all exact gradient of top gradient points and weighted aggregated datapoints of this sample datapoints.

How do you do it, sort the data based of the gradient, select the top 20% of the data points and from the remaining 80% only select the certain percentage(10%).i.e. if you have 1000 datapoints then you would have (200 + 80) i.e. 280 datapoints to train on, just the 80 datapoints would have weightage of (1-b)/a, 8 in above case, as to have the representation of the missed out points. So, basically for each split if which could be a mixture of points from any of the set, just while doing node calculation (gain etc.) if the datapoint belongs to the sampled set then it would get multiplied with the weight. High gradient points are prioritised and that is why they not sampled.

**Nan value treatment is same as XGBOOST.**

**Which one to use when gradient boosting, XGBOOST and LIGHTGBM?**

LightGBM is very handy on large data as it is faster and memory wise with good accuracy with better categorical feature support. Where as the XGBOOST is slower with high memory usage but often with highest accuracy especially on smaller datasets when properly tuned.

**What is Parametric and non-parametric models?**

1. GIT:

Version control software which is used to keep the different versions of the software, it takes the snapshots of the software and maintains it.

There are two types:

1. Distributed
2. Centralized

Advantages:

* Bug fixing
* Version control
* Non-linear development
* Collaborative development

Example to understand is google sheet which takes a snapshot after updates and gives you the ability to go back to any of the version.

Git-Bash – command line tool.

Git – GUI tool for the same thing.

Main Hierarchy:

1. Python.
2. VS Code, PyCharm, GIT, Streamlit. (More practical)
3. Stats
4. ML
5. DL
6. GEN AI

Page settings:

Margines: 2 X 2

Indentations

0.25-0.50-0.75-0.75

Spacing vertical: before after is 3-6, Encase of both specifying higher one is taken.