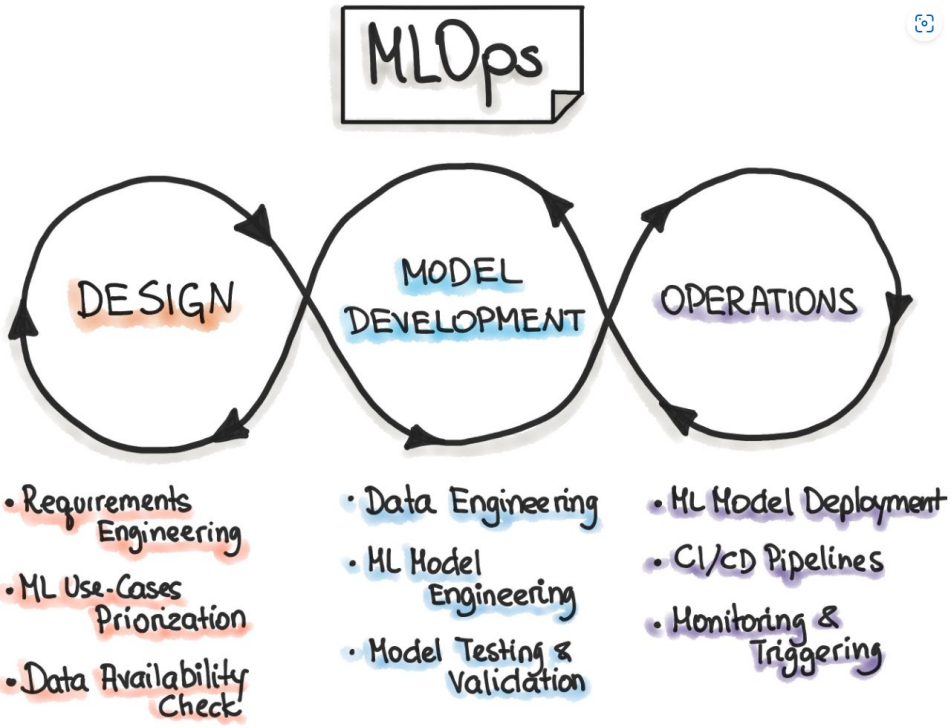
MLOps

MLOps (Machine Learning Operations) is the practice of applying DevOps principles and practices to machine learning projects. It involves the integration of machine learning development, deployment, and management into the overall software development process.

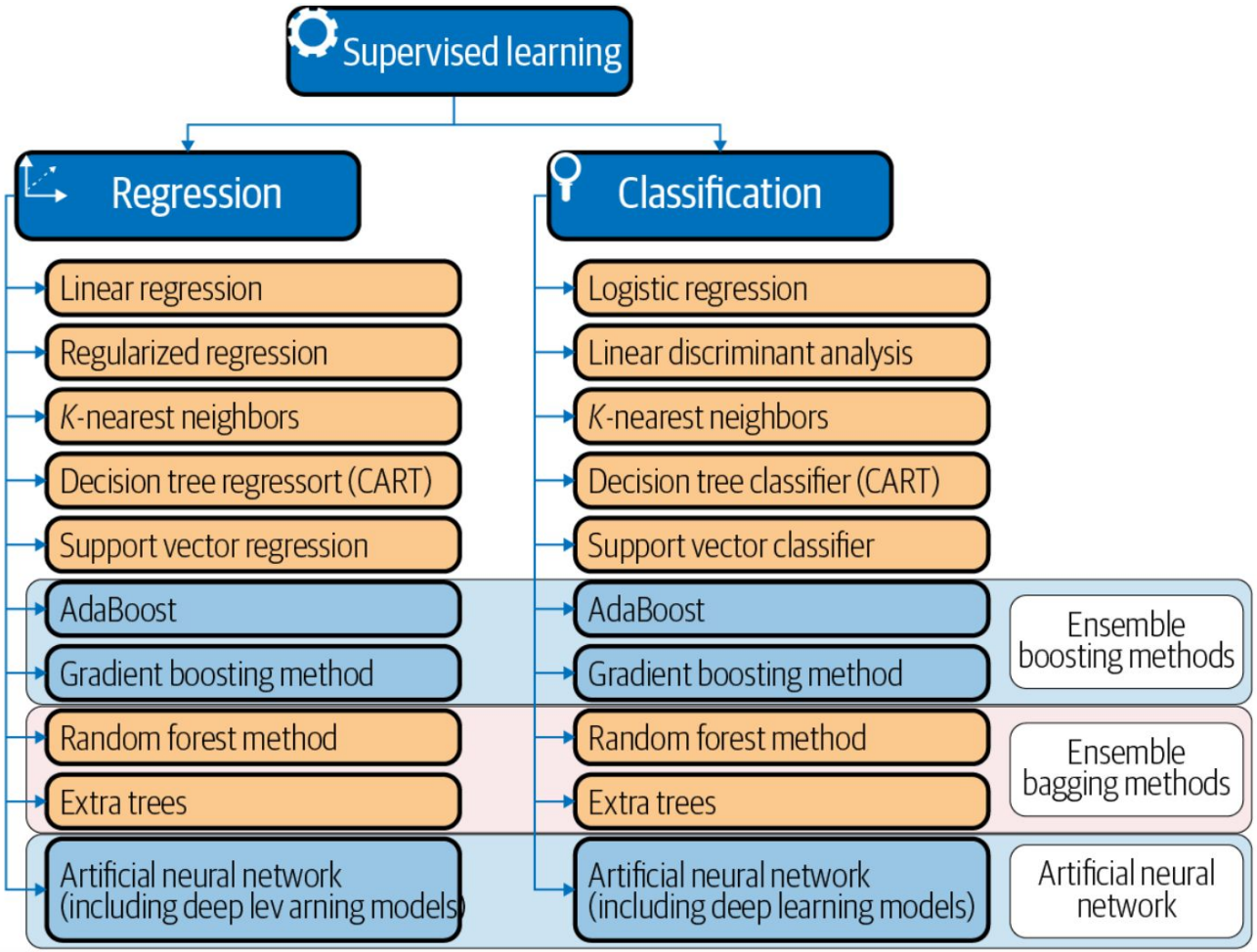


MLOps Prozess im Überblick

- **The Complete MLOps Study Roadmap:**

1. Understand the basics of machine learning Algorithms.

1.1. Supervised learning algorithms:



- **Regression:**

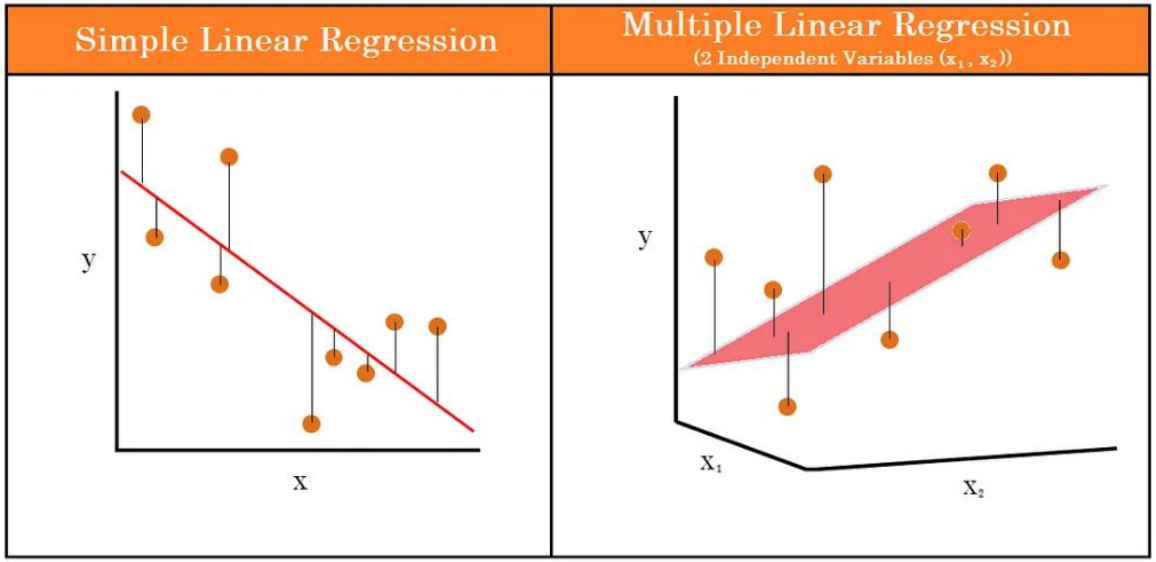
1. **Linear Regression:** attempts to model the relationship between two variables by fitting a **linear** equation to observed data. One variable is considered to be an explanatory variable or an independent variable, and the other is considered to be a dependent variable.

   - **Mathematical Interpretation:**

     - **Simple Linear Regression:** $y = w_o + w_1 x$

     - y is the output variable. It is also called the *target variable* in machine learning, or the *dependent variable* in statistical modeling. It represents the continuous value that we are trying to predict.
     - x is the input variable. In machine learning, x is referred to as the *feature,* while in statistics, it is called the *independent variable*. It represents the information given to us at any given time.
     - $w0$ is the *bias term* or y-axis intercept.
     - $w1$ is the *regression coefficient* or scale factor. In classical statistics, it is the equivalent of the slope on the best-fit straight line that is produced after the linear regression model has been fitted.
     - wi are called *weights* in general.

- **Multiple Linear Regression:** $y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n$
  - *Where xi is the i-th feature with its own wi weight.*



- **model parameters:**
  - **The coefficients:** represent the strength and direction of the relationship between each independent variable (also known as a feature or predictor) and the dependent variable (also known as the target or output).
  - **The intercept:** represents the point at which the linear regression line crosses the y-axis.

- **Here is a summary of how linear regression works:**

1. **Data Preparation:** The first step is to prepare the data by transforming the independent variables into a suitable format and splitting the data into training and testing sets.
2. **Model Definition:** A linear regression model is defined by an equation of the form:

$$y = \beta 0 + \beta 1x1 + \beta 2x2 + \ldots + \beta nxn$$

where y is the target variable, x1, x2, ..., xn are the independent variables, and β0, β1, β2, ..., βn are the coefficients (also known as weights) of the model.

3. **Model Training:** During training, the goal is to find the optimal values of the coefficients (β0, β1, β2, ..., βn) that minimize the difference between the predicted values and the actual values of the target variable. This is typically done using a optimization algorithm such as gradient descent, which updates the coefficients in the direction of the steepest decrease in the loss function.

4. **Model Evaluation:** After the model has been trained, its performance is evaluated on the test data using metrics such as mean squared error (MSE) for regression problems and accuracy for classification problems.

5. **Model Deployment:** Once the model has been trained and evaluated, it can be deployed for making predictions on new data. In linear regression, the predictions are made by plugging in the values of the independent variables into the equation of the model and solving for the target variable.

2. **Regularized Regression:** is a technique used to prevent overfitting in a model by adding a penalty term to the loss function. This penalty term, called a regularization term by reducing the complexity of the final estimated model.

- **Ridge Regression or L2 Regularization:** adds a term to the cost function that is proportional to the square of the weight coefficients.

$$J(w_1, w_2, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda * (w_1^2 + w_2^2)$$

The power of Ridge Regression is that it minimize the RSS by enforce the W coefficients to be lower, but it **does not** enforce them to be zero.

- **Lasso Regression or L1 Regularization:** It adds a term to the cost function that is proportional to the absolute value of the weight coefficients

$$J(w_1, w_2, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda * (|w_1| + |w_2|)$$

A subset of the coefficients are forced to be precisely zero.

- **Elastic Net Regularization:** This is a combination of both L1 and L2 regularization.

$$J(w_1, w_2, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda * \alpha * (|w_1| + |w_2|) + \lambda * (1 - \alpha) * (w_1^2 + w_2^2)$$

- **When to use L1, L2 or Elastic Net?**

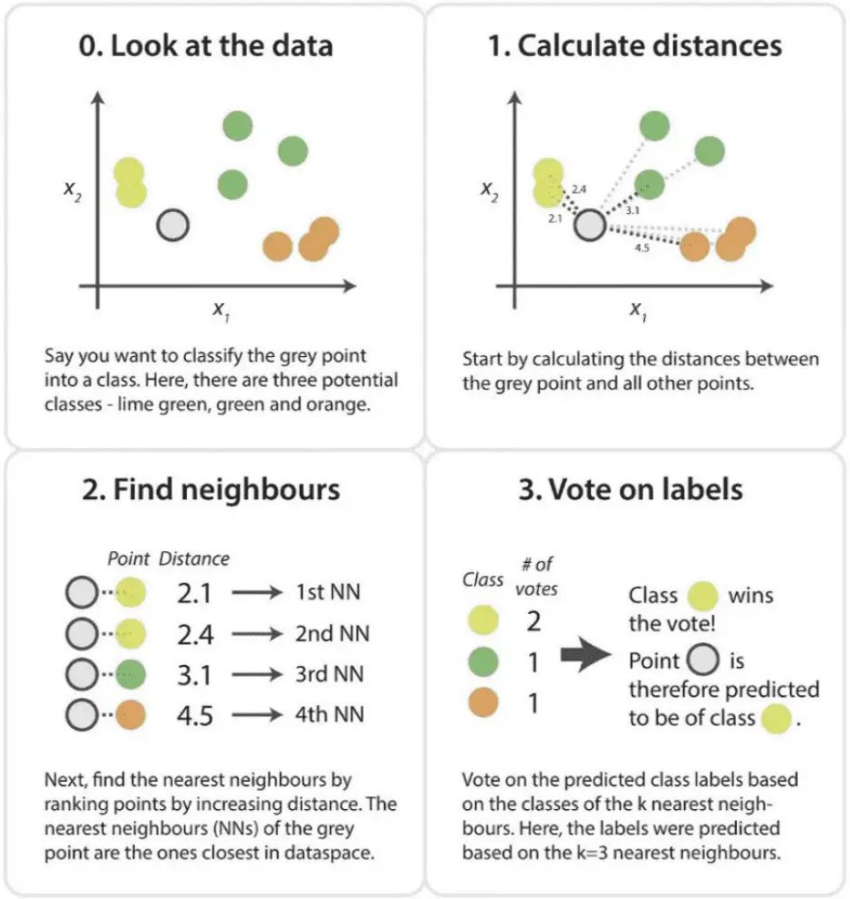
In many scikit-learn models L2 is the default (see LogisticRegression and SupportVectorMachines). This is for a reason: L1 tends to shrink some of the weight coefficients to zero, which means the features are removed from te model. So L1 regularization is more useful for feature selection. To really prevent overfitting, L2 might be the better choice, because it does not set any of the weight coefficients to zero.

Elastic Net regularization is a good choice when you have correlated features and you want to balance the feature selection and overfitting prevention. It's also useful when you're not sure whether L1 or L2 regularization would be more appropriate for your data and model.

In general, L2 regularization is recommended when you have a large number of features and you want to keep most of them in the model, and L1 regularization is recommended when you have a high-dimensional dataset with many correlated features and you want to select a subset of them.

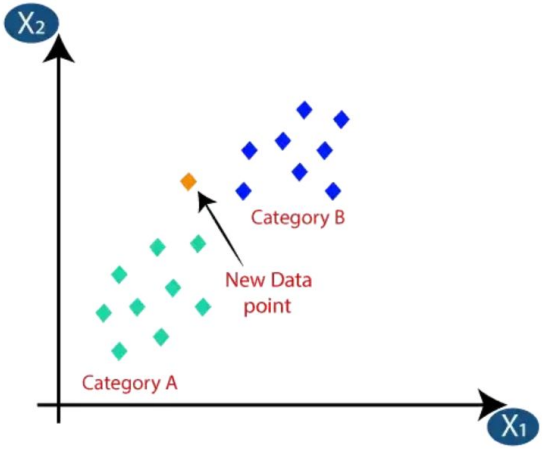
- **model parameters:**
  - **regularization parameter:** often denoted as lambda or alpha, which controls the strength of the regularization term. A larger value of lambda or alpha corresponds to stronger regularization and a smaller value corresponds to weaker regularization.

- **K-Nearest Neighbor:** used for both regression and classification. KNN tries to predict the correct class for the test data by calculating the distance between the test data and all the training points. Then select the K number of points which is closest to the test data. The KNN algorithm calculates the probability of the test data belonging to the classes of ‘K’ training data and class holds the highest probability will be selected. In the case of regression, the value is the mean of the ‘K’ selected training points.

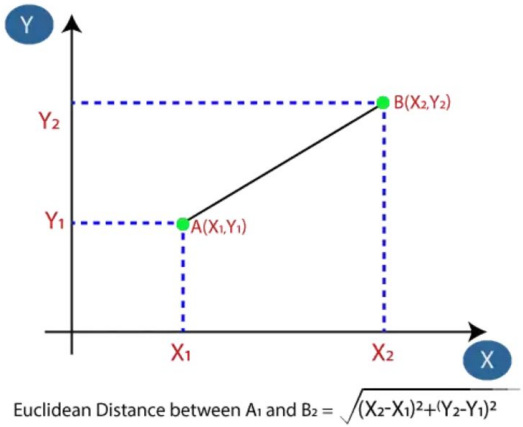


- **How does K-NN work?**

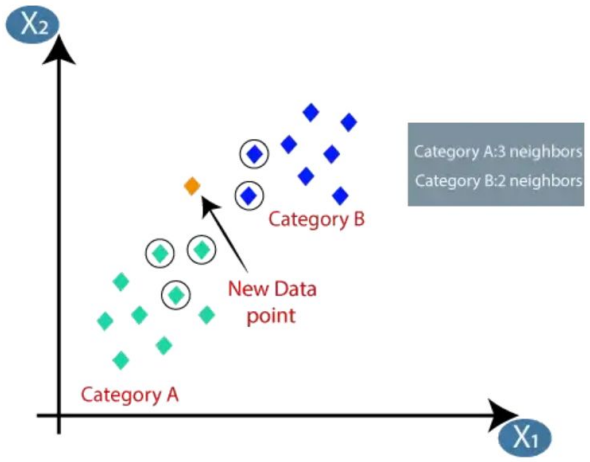
  - Suppose we have a new data point and we need to put it in the required category. Consider the below image:



  - Firstly, we will choose the number of neighbors, so we will choose the k=5.
  - Next, we will calculate the Euclidean distance between the data points. The Euclidean distance is the distance between two points.



- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B.



- the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

- **how to select the optimal K value?**

To select the optimal value of K, a range of values for K are tested, typically starting from small values such as 1 or 3 and increasing to larger values such as 10 or 20. The performance of the model is then evaluated for each value of K using k-fold cross-validation, and the value of K that results in the best performance is chosen as the optimal value.

Another way to select K is the elbow method, that is looking at the plot of the error rate versus the value of K, the elbow point is the optimal K where the rate of decrease in error rate starts to level off.

- **model parameters:**
  - **k_numbers** : this parameter controls the number of nearest neighbors that the algorithm uses to make a prediction.
  - **Distance metric:** The method used to calculate the distance between data points.

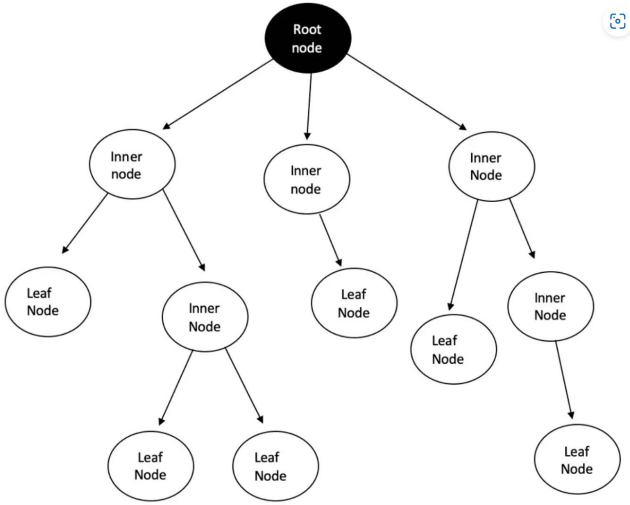
- Euclidean distance: This is the most commonly used distance metric in KNN. It is the straight-line distance between two points in Euclidean space. It's calculated by taking the square root of the sum of the squares of the differences between the coordinates of the two points.
- Manhattan distance: Also known as the "taxi cab" distance, this metric is calculated by summing the absolute differences of the coordinates of the two points along each dimension. It's calculated as the sum of the absolute difference between the coordinates of two points.
- Minkowski distance: This is a generalization of both Euclidean and Manhattan distance and is parameterized by a value "p". When p = 2, it's equivalent to Euclidean distance, when p=1 it's equivalent to Manhattan distance.
- Chebyshev distance: This metric is calculated as the maximum absolute difference between the coordinates of the two points. It's an upper bound on the other distance metrics.
- Cosine similarity: This metric is used when the data is in the form of vectors. It measures the cosine of the angle between two vectors and ranges between [-1,1]. A cosine similarity of 1 means that the vectors point in the same direction, while a cosine similarity of -1 means that they point in opposite directions.
- Jaccard distance: This metric is used when the data is in the form of binary vectors. It measures the dissimilarity between two sets and ranges between [0,1]. A Jaccard distance of 0 means that the sets have no elements in common, while a Jaccard distance of 1 means that the sets are identical.

- **Weighting method:** used to assign weights to the k nearest neighbors when making a prediction.

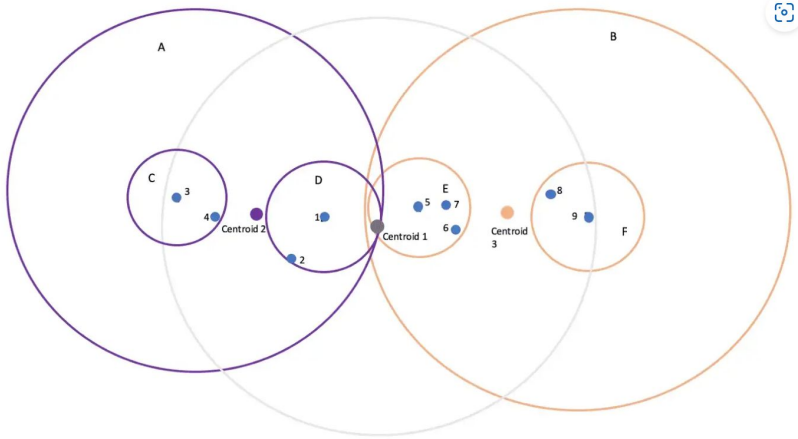
  - Uniform weighting: In this method, each of the k nearest neighbors is given the same weight when making a prediction. This is the simplest weighting method, but it may not be optimal for all datasets.
  - Distance-weighted: In this method, the weight of each of the k nearest neighbors is inversely proportional to its distance from the test point. This means that the closer a neighbor is to the test point, the higher its weight will be. This weighting method gives more importance to the closest neighbors, and it's more suitable for datasets with a lot of noise.
  - Kernel weighting: In this method, a kernel function is used to assign weights to the k nearest neighbors. A kernel function is a function that measures the similarity between two points in a high-dimensional feature space. Common kernel functions include the Gaussian kernel, the polynomial kernel, and the radial basis function kernel. This weighting method is useful for datasets with non-linear decision boundaries.
  - Adaptive weighting: In this method, the weights of the k nearest neighbors are adaptively adjusted based on the density of the neighborhood. This weighting method is useful for datasets with varying densities.

- **algorithm:** {‘auto’, ‘ball_tree’, ‘kd_tree’, ‘brute’}, default=’auto’}

- Brute force: This is the simplest and most straightforward algorithm, in which the distance between the test point and all the points in the training set are calculated and the k nearest neighbors are selected. This algorithm has a time complexity of O(Nd), where N is the number of points in the training set and d is the number of dimensions of the data.
- K-D Tree: This algorithm uses a k-dimensional tree data structure to partition the data into smaller subsets and quickly find the k nearest neighbors. The time complexity of this algorithm is O(log N), which is significantly faster than the brute force algorithm.
- Ball tree: This algorithm is similar to the K-D tree algorithm, but it uses a ball tree data



k-Dimensional Tree (kd tree)



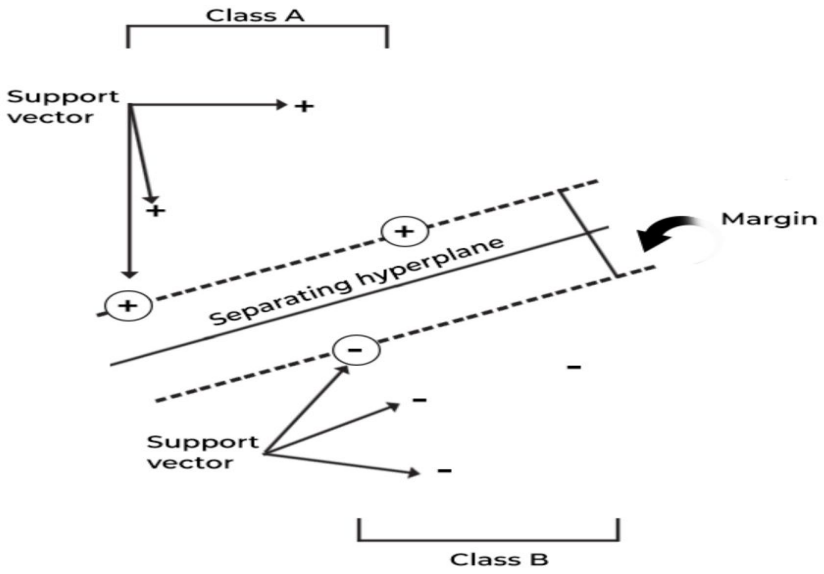
Ball Tree

- **Here is a summary of how K-Nearest Neighbor (KNN) works:**

1. Data Preparation: The first step is to prepare the data by transforming the independent variables into a suitable format and splitting the data into training and testing sets.
2. Model Definition: In KNN, there is no explicit model definition. Instead, the algorithm simply stores the training samples in memory and uses them to make predictions.
3. Distance Measurement: During prediction, the algorithm calculates the distance between the test sample and each of the training samples. The distance metric used can be any metric that measures the difference between two samples, such as Euclidean distance, Manhattan distance, or Cosine similarity.
4. Selection of K: The next step is to select the value of K, which determines the number of nearest neighbors used for prediction. In general, a larger value of K results in a smoother decision boundary and reduces the impact of outliers, while a smaller value of K results in a more complex decision boundary and increases the sensitivity to outliers.
5. Prediction: After the distances have been calculated and K has been selected, the prediction for the test sample is based on the average of the K nearest training samples. For classification problems, the prediction is the majority class among the K nearest neighbors. For regression problems, the prediction is the average value of the target variable among the K nearest neighbors.
6. Model Evaluation: After the predictions have been made, the performance of the model is evaluated on the test data using metrics such as accuracy for classification problems and mean squared error (MSE) for regression problems.

- **Support Vector Machine (SVM):** It can be applied for both regression and classification problems but is most commonly used for classification. It is a supervised learning algorithm which identifies the best hyperplane to divide the dataset.

    - **Support Vectors**: the points which are closest to the hyperplane.
    - **hyperplane**: the decision boundary between different classes.
    - **Margin** : the distance between the hyperplane and the nearest data point from either side.
    - **Kernel** :a mathematical function used to transform input data into a different form. Common kernel functions include linear, nonlinear, polynomial, etc.



- **How Does a Support Vector Machine Work?**

The basic idea behind an SVM is to find a hyperplane that maximally separates the different classes in the data. This is done by mapping the input data into a higher-dimensional space and then finding the hyperplane that maximally separates the classes in this space. The distance from the hyperplane to the closest data points from each class is called the margin. The goal is to choose a hyperplane with the largest possible margin, as this will lead to the most robust classification.

- **Mathematically:**

Given a set of training examples {(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)} where x_i is a feature vector and y_i is a binary label (-1 or 1), the goal is to find a hyperplane defined by the equation:

$w^T x + b = 0$

where w is a weight vector and b is a bias term.

The decision boundary of this hyperplane is determined by the sign of $w^T x + b$. If $w^T x + b > 0$, then the example is classified as positive; otherwise, it is classified as negative.

To maximize the margin, which is the distance between the hyperplane and the closest examples from each class, the following optimization problem is solved:

minimize $(1/2) \|w\|^2$

subject to $y_i (w^T x_i + b) >= 1$ for i = 1, 2, ..., n

- **Types of Support Vector Machines:**

1. **Linear SVM:** Linear SVMs are used for linearly separable data, where the decision boundary is a straight line or a hyperplane. In this case, the optimization problem is to find the hyperplane with the largest margin.
2. **Non-linear SVM:** Non-linear SVMs are used for non-linearly separable data, where the decision boundary is not a straight line or a hyperplane. In this case, a kernel trick is used to map the input data into a higher-dimensional space where it becomes linearly separable.
3. **Soft-Margin SVM:** Soft-Margin SVM is used when the data is not perfectly linearly separable. It introduce a slack variable ϵi for each data point, which allows some data points to be on the wrong side of the margin or even the wrong side of the decision boundary. This can be controlled by a parameter C, which determines the trade-off between maximizing the margin and allowing misclassifications.
4. **Multi-class SVM:** Multi-class SVMs are used for multi-class classification problems, where the goal is to classify the input data into more than two classes. There are different ways to extend binary SVM to multi-class classification, such as one-vs-all or one-vs-one.
5. **Support Vector Regression (SVR):** Support Vector Regression (SVR) is a type of SVM that is used for regression tasks, rather than classification tasks. In this case, the goal is to find a hyperplane that best approximates the underlying function that generated the data.

- **Model parameters:**

  - **C**: The C parameter is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the error. A larger value of C will lead to a smaller margin but fewer misclassifications, while a smaller value of C will lead to a larger margin but more misclassifications.
  - **Gamma**: The gamma parameter is used in the RBF kernel and controls the width of the Gaussian function. A larger value of gamma will lead to more complex decision boundary, while a smaller value of gamma will lead to a wider and simpler decision boundary.
  - **Degree**: The degree parameter is used in the polynomial kernel and controls the degree of the polynomial. A larger value of degree will lead to a more complex decision boundary, while a smaller value of degree will lead to a simpler decision boundary.
  - **Epsilon**: The epsilon parameter is used in the Support Vector Regression (SVR) and controls the width of the margin. A smaller value of epsilon will lead to a narrower margin and more accurate predictions, while a larger value of epsilon will lead to a wider margin and less accurate predictions.
  - **Kernel**: is used to map the input data into a higher-dimensional space where it becomes linearly separable. There are several types of kernel functions that can be used in SVM models:
    - **Linear:** The linear kernel is the simplest kernel and is used when the data is already linearly separable. It is defined as the dot product of two input vectors.
    - **Polynomial:** The polynomial kernel is used when the data is not linearly separable. It is defined as (gamma * dot product + coef0)^degree, where gamma, coef0 and degree are parameters that can be adjusted to improve performance.
    - **Radial basis function (RBF):** The RBF kernel is also used when the data is not linearly separable. It is defined as exp(-gamma * ||x-y||^2), where gamma is a parameter that controls the width of the Gaussian function and x and y are input vectors.
    - **Sigmoid:** The sigmoid kernel is defined as tanh(gamma * dot product + coef0), where gamma, coef0 are parameters that can be adjusted to improve performance.

  - **The main difference between polynomial and RBF kernel:** the polynomial kernel creates a boundary that is shaped like a polynomial equation, and the RBF kernel creates a boundary that is shaped like a bell curve. The RBF is more flexible and works well with most datasets, but polynomial might be a good choice in case the data can be separated by a polynomial boundary.

- **Here is a summary of how Support Vector Machine (SVM) works:**

The main idea behind SVM is to find a hyperplane that best separates the data into classes. A hyperplane is a line or a higher dimensional equivalent, such as a plane, that separates the data into two classes. The hyperplane that provides the largest separation between the classes is known as the maximum margin hyperplane. The margin is defined as the distance between the hyperplane and the closest data points to the hyperplane, known as support vectors.

The support vectors are the key to finding the maximum margin hyperplane. They are the data points that determine the position and orientation of the hyperplane. By finding the support vectors, SVM can classify new data points based on their proximity to the hyperplane.

In the case of a non-linear problem, SVM transforms the original data into a higher dimensional space, where a linear separation is possible. This process is known as kernel trick. The most commonly used kernel functions are radial basis function (RBF) and polynomial.

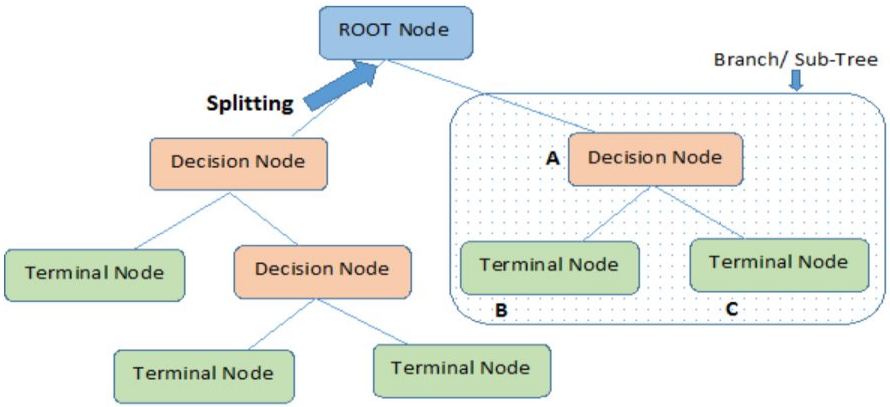
In the case of classification problems, SVM outputs a binary classifier. In the case of regression problems, SVM outputs a real-valued prediction. In both cases, SVM can be regularized to avoid overfitting by adding a hyperparameter, known as C, that controls the trade-off between achieving a low training error and a low testing error.

- **Decision Tree:** are now widely used in many applications for predictive modeling, including both classification and regression. Sometimes decision trees are also referred to as **CART**, which is short for Classification and Regression Trees.

  - **Types of Decision Trees**

  - **Categorical Variable Decision Trees**: This is where the algorithm has a categorical target variable.
  - **Continuous Variable Decision Trees**: In this case the features input to the decision tree (e.g. qualities of a house) will be used to predict a continuous output (e.g. the price of that house).

  - **How does it work?**
    The basic structure of the decision tree: Every tree has a root node, where the inputs are passed through. This root node is further divided into sets of decision nodes where results and observations are conditionally based. The process of dividing a single node into multiple nodes is called splitting. If a node doesn't split into further nodes, then it's called a leaf node, or terminal node. A subsection of a decision tree is called a branch or sub-tree



Example of a Decision Tree

- **NOTE:** There is also another concept that is quite opposite to splitting. If there are ever decision rules which can be eliminated, we cut them from the tree. This process is known as **pruning**, an is useful to minimize the complexity of the algorithm.

- **How To Create a Decision Tree ?**

The main goal of decision trees is to make the best splits between nodes which will optimally divide the data into the correct categories.

- The commonly used techniques to split:
- **Gini Impurity:** If all elements are correctly divided into different classes, the division is considered to be pure. The Gini impurity is used to gauge the likelihood that a randomly chosen example would be wrongly classified by a certain node. It is known as an "impurity" measure since it gives us an idea of how the model differs from a pure division.

  The degree of the Gini impurity score is always between 0 and 1, where 0 denotes that all elements belong to a certain class (or the division is pure), and 1 denotes that the elements are randomly distributed across various classes. A Gini impurity of 0.5 denotes that the elements are distributed equally into some classes.

$$Gini = 1 - \sum_{i=1}^{n} (p_i)^2$$

Where p_i is the probability of a particular element belonging to a specific class.

Class	Var 1	Var 2
A	0	33
A	0	54
A	0	56
A	0	42
A	1	50
B	1	55
B	1	31
B	0	-4
B	1	77
B	0	49

Gini Index Example:

- The baseline of the split for *Var1*: *Var1* has 4 instances (4/10) equal to 1 and 6 instances (6/10) equal to 0.
- For *Var1* == 1 & *Class* == *A*: 1 / 4 instances have class equal to *A*.
- For *Var1* == 1 & *Class* == *B*: 3 / 4 instances have class equal to *B*.
- Gini Index here is 1-((1/4)^2 + (3/4)^2) = 0.375
- For *Var1* == 0 & *Class* == *A*: 4 / 6 instances have class equal to *A*.
- For *Var1* == 0 & *Class* == *B*: 2 / 6 instances have class equal to *B*.
- **Gini Index** here is 1-((4/6)^2 + (2/6)^2) = 0.4444
- We then weight and sum each of the splits based on the baseline / proportion of the data each split takes up.
- 4/10 * 0.375 + 6/10 * 0.444 = 0.41667

- **Information Gain:** Information Gain tells us how important the attribute is. Since Decision Tree construction is all about finding the right split node that assures high accuracy, Information Gain is all about finding the best nodes that return the highest information gain. This is computed using a factor known as Entropy. The more the disorganization is, the more is the entropy. When the sample is wholly homogeneous, then the entropy turns out to be zero, and if the sample is partially organized, say 50% of it is organized, then the entropy turns out to be one.

  - Calculate the entropy of the output attribute (before the split) using the formula,

$$E(s) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

Here, p is the probability of success and q is the probability of failure of the node. Say, out of the 10 data values, 5 pertain to *True* and 5 pertain to *False*, then *c* computes to 2, *p_1* and *p_2* compute to ½.

  - Using the above two values, calculate the Information Gain or the decrease in entropy by subtracting the entropy of each attribute from the total entropy before the split,

$$Gain(T, X) = Entropy(T) - Entropy(T, X)$$

  - Choose the attribute that has the highest information gain as the split node.

  - Calculate the entropy of all the input attributes using the formula,

$$E(T, x) = \sum_{c \in X} P(c) E(c)$$

T is the output attribute,

X is the input attribute,

P(c) is the probability w.r.t the possible data point present at X, and

E(c) is the entropy w.r.t 'True' pertaining to the possible data point.

Assume an input attribute (priority) where there are two possible values mentioned, *low* and *high*. With respect to *low*, there are 5 data points associated, out of which, 2 pertain to *True* and 3 pertain to *False*. With respect to *high*, the remaining 5 data points are associated, wherein 4 pertain to *True* and 1 pertains to *False*. Then E(T, X) would be,

$$E(T, x) = \frac{5}{10} * E(2, 3) + \frac{5}{10} * E(4, 1)$$

In E(2, 3), p is 2, and q is 3.

In E(4, 1), p is 4, and q is 1.

- A leaf node is the one that has no entropy, or when the entropy is zero. No further splitting is done on a leaf node.

- model parameters:

- **criterion**: This parameter is used to measure the quality of the split. The default value for this parameter is set to “Gini”. If you want the measure to be calculated by entropy gain, you can change this parameter to “entropy”.
- **splitter**: This parameter is used to choose the split at each node. If you want the sub-trees to have the best split, you can set this parameter to “best”. We can also have a random split for which the value “random” is set.
- **max-depth**: This is an integer parameter through which we can limit the depth of the tree. The default value for this parameter is set to None.
- **min_samples_split**: This parameter is used to define the minimum number of samples required to split an internal node.
- **max_leaf_nodes**: The default value of max_leaf_nodes is set to None. This parameter is used to grow a tree with max_leaf_nodes in best-first fashion.

The criterion parameter in a decision tree regressor controls the function used to evaluate the quality of a split when building the tree. The two most commonly used criteria in decision tree regression are:

- Mean Squared Error (MSE): The MSE criterion measures the average squared difference between the predicted values and the true values for the samples in a given node. The goal is to minimize the MSE by selecting the best feature and threshold value for the split.
- Mean Absolute Error (MAE): The MAE criterion measures the average absolute difference between the predicted values and the true values for the samples in a given node. The goal is to minimize the MAE by selecting the best feature and threshold value for the split.

These criterion are used to split the data into subsets and to predict the target variable. The decision tree algorithm will continue to split the data recursively until a stopping criterion is met, such as a maximum tree depth or a minimum number of samples in a leaf node, then it will be used to make predictions for new samples by traversing the tree.

- **Here is a summary of how Decision tree works:**

A Decision Tree is a tree-based machine learning algorithm used for both classification and regression problems. It works by recursively splitting the data into subsets based on the values of the input features, creating a tree-like structure. Each internal node of the tree represents a test on one of the input features, and each leaf node represents a prediction. The prediction can either be a class label in the case of a classification problem or a continuous value in the case of a regression problem.

The process of constructing a decision tree starts with the root node, which represents the entire dataset. At each internal node, the algorithm selects the feature that provides the best split of the data into two subsets. The best split is determined by a criterion such as information gain, gain ratio, or Gini impurity, which measures the quality of the split. Once the best feature is selected, the data is divided into two subsets based on its values, and the process is repeated for each subset. The process continues until a stopping criterion is met, such as a minimum number of samples required in a leaf node, a maximum tree depth, or no further improvement in the quality of the splits.

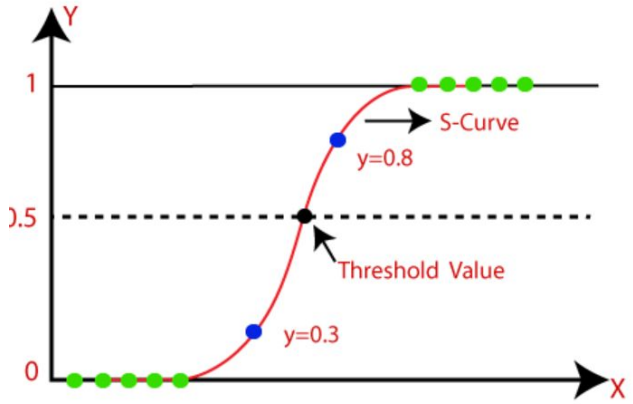
After the tree is constructed, it can be used to make predictions for new data points. To make a prediction, the algorithm starts at the root node and follows the branches of the tree based on the values of the input features, until it reaches a leaf node. The prediction is made based on the class label or continuous value assigned to the leaf node.

In practice, decision trees can be prone to overfitting, especially when the tree is deep and complex. To mitigate this, various techniques such as pruning, ensemble methods, and limiting the tree depth can be used.

- **Logistic regression:** is a classification algorithm. It is used to predict a binary outcome based on a set of independent variables.
  - A **binary outcome:** is one where there are only two possible scenarios—either the event happens (1) or it does not happen (0). it gives the probabilistic values which lie between 0 and 1.
  - **Independent variables:** are those variables or factors which may influence the outcome (or dependent variable).

  - The independent variables can fall into any of the following categories:
  1. **Continuous**: such as temperature in degrees Celsius or weight in grams.
  2. **Discrete, ordinal**—that is, data which can be placed into some kind of order on a scale. For example, if you are asked to state how happy you are on a scale of 1-5.
  3. **Discrete, nominal**—that is, data which fits into named groups which do not represent any kind of order or scale. For example, eye color may fit into the categories “blue”, “brown”, or “green”.

- In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
- The curve from the logistic function indicates the likelihood of something.



- Logistic Function (Sigmoid Function):
  - The sigmoid function is a mathematical function used to map the predicted values to probabilities.
  - It maps any real value into another value within a range of 0 and 1.
  - In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1.

- Assumptions for Logistic Regression:
  - The dependent variable must be categorical.
  - The independent variable should not have multi-collinearity.

- Hypothesis Representation:

  - Using *linear regression* we used a formula of the hypothesis:
    $h\Theta(x) = \beta_0 + \beta_1 X$
  - For logistic regression:
    $\sigma(Z) = \sigma(\beta_0 + \beta_1 X)$

  $Z = \beta_0 + \beta_1 X$

  $h\Theta(x) = \text{sigmoid}(Z)$

  i.e. $h\Theta(x) = 1/(1 + e^{-(\beta_0 + \beta_1 X)})$

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$h\theta(X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

- **Types of logistic regression:**

1. **Binary logistic regression** is the statistical technique used to predict the relationship between the dependent variable (Y) and the independent variable (X), where the dependent variable is binary in nature. For example, the output can be Success/Failure, 0/1 , True/False, or Yes/No. This is the type of logistic regression that we've been focusing on in this post.

2. **Multinomial logistic regression** is used when you have one categorical dependent variable with two or more unordered levels (i.e two or more discrete outcomes). It is very similar to logistic regression except that here you can have more than two possible outcomes. For example, let's imagine that you want to predict what will be the most-used transportation type in the year 2030. The transport type will be the dependent variable, with possible outputs of train, bus, tram, and bike (for example).

3. **Ordinal logistic regression** is used when the dependent variable (Y) is ordered (i.e., ordinal). The dependent variable has a meaningful order and more than two categories or levels. Examples of such variables might be t-shirt size (XS/S/M/L/XL), answers on an opinion poll (Agree/Disagree/Neutral), or scores on a test (Poor/Average/Good).

- Model parameters:
  - The coefficients represent the contribution of each feature/predictor variable to the predicted probability of the target variable being 1.
  - The intercept represents the baseline probability of the target variable being 1 when all predictors have a value of 0.

- **Here is a summary of how Logistic regression works:**

Logistic Regression is a type of generalized linear model that is used for binary classification problems. It is used to model the relationship between a dependent variable and one or more independent variables. In binary classification, the dependent variable is a binary variable that represents the class label, and the independent variables are the input features.

The goal of logistic regression is to find a model that estimates the probability of the positive class given the input features. The model is represented by a linear combination of the input features, weighted by coefficients. The linear combination is transformed by the logistic function, which maps the result to a value between 0 and 1, which represents the probability of the positive class.

The logistic function is defined as:

$p = 1 / (1 + \exp(-z))$

where p is the predicted probability of the positive class, and z is the linear combination of the input features:

$z = b0 + b1 * x1 + b2 * x2 + ... + bn * xn$

where b0, b1, b2, ..., bn are the coefficients that represent the weight of each input feature, and x1, x2, ..., xn are the input features.

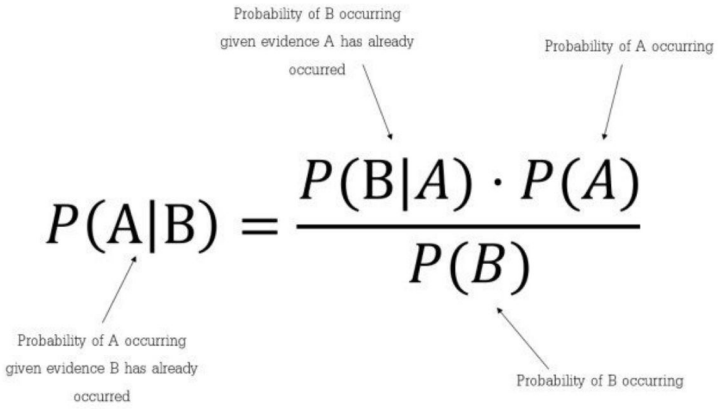
The coefficients are estimated using maximum likelihood estimation, which seeks to find the values of the coefficients that maximize the likelihood of the observed data. The likelihood is defined as the product of the probabilities of the observed data given the input features, calculated using the logistic regression model.

Once the coefficients are estimated, the logistic regression model can be used to make predictions for new data points. To make a prediction, the input features are plugged into the logistic regression model to calculate the predicted probability of the positive class. If the predicted probability is greater than or equal to 0.5, the model predicts the positive class, otherwise, it predicts the negative class.

- **Naive Bayes:** is a probabilistic machine learning algorithm based on the **Bayes Theorem**, used in a wide variety of classification tasks.

  - **Bayes' Theorem:** is a simple mathematical formula used for calculating conditional probabilities.

  - **Conditional probability** is a measure of the probability of an event occurring given that another event has occurred.



  - The formula tells us: how often A happens *given that B happens*, written **P(A|B)** also called posterior probability, When we know: how often B happens *given that A happens*, written **P(B|A)** and how likely A is on its own, written **P(A)** and how likely B is on its own, written **P(B).**

Shopping Example

Problem statement: To predict whether a person will purchase a product on a specific combination of day, discount, and free delivery using a Naive Bayes classifier.

See a small sample data set of 30 rows, with 15 of them, as shown below:

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

customer_id	first_name	last_name	birth_date	phone	address	city	state	points
1	Babara	MacCaffrey	1986-03-28	781-932-9754	0 Sage Terrace	Waltham	MA	2273
2	Ines	Brushfield	1986-04-13	804-427-9456	14187 Commercial Trail	Hampton	VA	947
3	Freddi	Boagey	1985-02-07	719-724-7869	251 Springs Junction	Colorado Springs	CO	2967
4	Ambur	Roseburgh	1974-04-14	407-231-8017	30 Arapahoe Terrace	Orlando	FL	457
5	Clemmie	Betchley	1973-11-07	NULL	5 Spohn Circle	Arlington	TX	3675
6	Elka	Twiddell	1991-09-04	312-480-8498	7 Manley Drive	Chicago	IL	3073
7	Ilene	Dowson	1964-08-30	615-641-4759	50 Lillian Crossing	Nashville	TN	1672

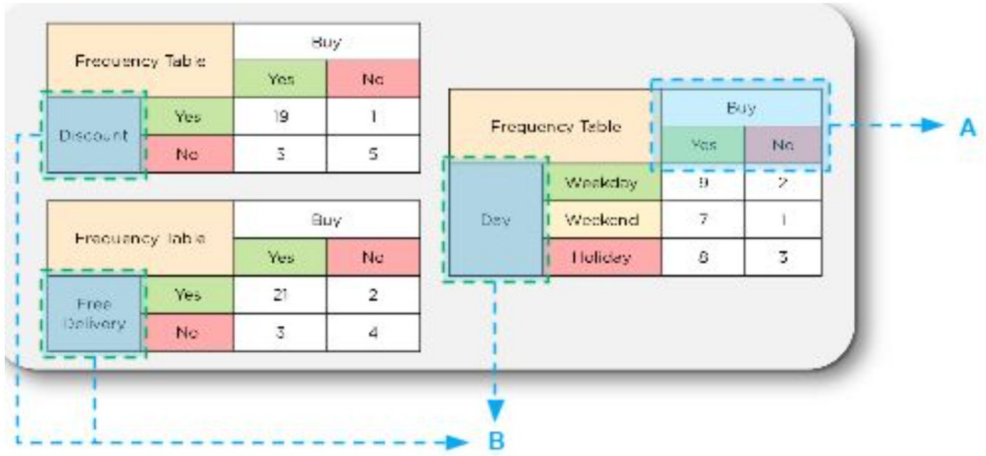
Based on the dataset containing the three input types—day, discount, and free delivery— the frequency table for each attribute is populated.

Frequency Table		Buy	
		Yes	No
Discount	Yes	19	1
	No	5	5

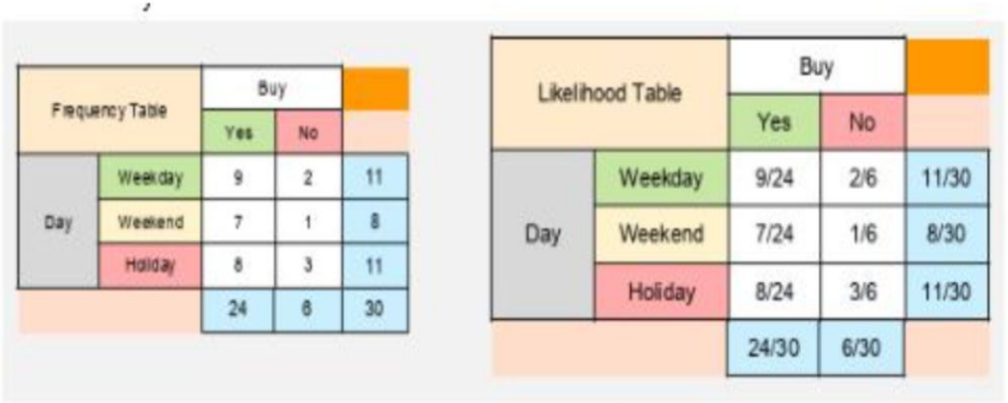
Frequency Table		Buy	
		Yes	No
Free Delivery	Yes	21	2
	No	3	4

Frequency Table		Buy	
		Yes	No
Day	Weekday	9	2
	Weekend	7	1
	Holiday	8	3

For Bayes theorem, let the event ‘buy’ be A and the independent variables (discount, free delivery, day) be B.



Let us calculate the likelihood for one of the “day” variables, which includes weekday, weekend, and holiday variables.



Frequency Table		Buy		
		Yes	No	
Day	Weekday	9	2	11
	Weekend	7	1	8
	Holiday	8	3	11
		24	6	30

Likelihood Table		Buy		
		Yes	No	
Day	Weekday	9/24	2/6	11/30
	Weekend	7/24	1/6	8/30
	Holiday	8/24	3/6	11/30
		24/30	6/30	

We get a total of:

11 weekdays

Eight weekends

11 holidays

The total number of days adds up to 30 days.

There are nine out of 24 purchases on weekdays

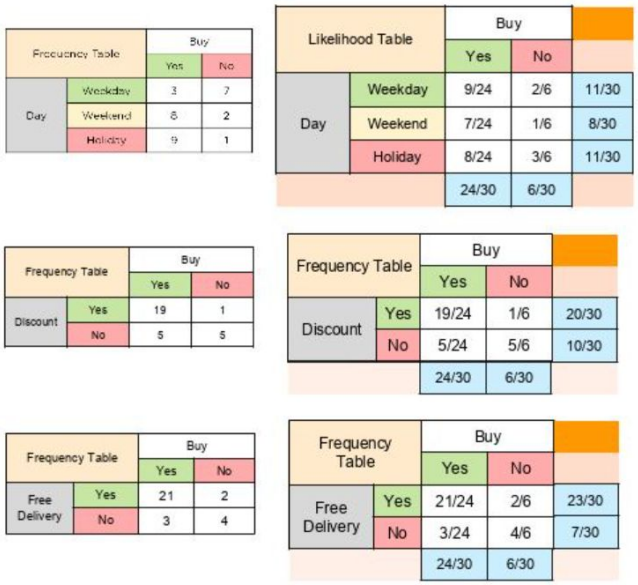
There are seven out of 24 purchases on weekends

There are eight out of 24 purchases on holidays

Based on the above likelihood table, let us calculate some conditional probabilities:

- P(B) = P(Weekday) = 11/30 = 0.37
- P(A) = P(No Buy) = 6/30 = 0.2
- P(B | A) = P(Weekday | No Buy) = 2/6 = 0.33
- P(A | B) = P(No Buy | Weekday) = P(Weekday| No Buy) * P(No Buy) / P(Weekday) = (0.33 * 0.2) / 0.37 = 0.18
- The probability of purchasing on the weekday = 11/30 or 0.37
- It means out of the 30 people who came into the store throughout the weekend, weekday, and holiday, 11 of those purchases were made on weekdays.
- The probability of not making a purchase = 6/30 or 0.2. There's a 20 percent chance that they're not going to make a purchase, no matter what day of the week it is.
- The probability of the weekday without a purchase = 0.18 or 18 percent. As the probability of (No | Weekday) is less than 0.5, the customer will most likely buy the product on a weekday.

See the likelihood tables for the three variables below:



Frequency Table		Buy	
		Yes	No
Day	Weekday	3	7
	Weekend	8	2
	Holiday	9	1

Likelihood Table		Buy		
		Yes	No	
Day	Weekday	9/24	2/6	11/30
	Weekend	7/24	1/6	8/30
	Holiday	8/24	3/6	11/30
		24/30	6/30	

Frequency Table		Buy	
		Yes	No
Discount	Yes	19	1
	No	5	5

Frequency Table		Buy		
		Yes	No	
Discount	Yes	19/24	1/6	20/30
	No	5/24	5/6	10/30
		24/30	6/30	

Frequency Table		Buy	
		Yes	No
Free Delivery	Yes	21	2
	No	3	4

Frequency Table		Buy		
		Yes	No	
Free Delivery	Yes	21/24	2/6	23/30
	No	3/24	4/6	7/30
		24/30	6/30	

The likelihood tables can be used to calculate whether a customer will purchase a product on a specific combination of the day when there is a discount and whether there is free delivery. Consider a combination of the following factors where B equals:

- Day = Holiday
- Discount = Yes
- Free Delivery = Yes

Let us find the probability of them not purchasing based on the conditions above.

A = No Purchase

Applying Bayes Theorem, we get P(A | B) as shown:

$$P(A|B) = P(\text{No Buy} \mid \text{Discount} = \text{Yes}, \text{Free Delivery} = \text{Yes}, \text{Day} = \text{Holiday})$$

$$= \frac{P(\text{Discount} = \text{Yes} \mid \text{No}) * P(\text{Free Delivery} = \text{Yes} \mid \text{No}) * P(\text{Day} = \text{Holiday} \mid \text{No}) * P(\text{No Buy})}{P(\text{Discount}=\text{Yes}) * P(\text{Free Delivery}=\text{Yes}) * P(\text{Day}=\text{Holiday})}$$

$$= \frac{(1/6) * (2/6) * (3/6) * (6/30)}{(20/30) * (23/30) * (11/30)}$$

$$= 0.178$$

Similarly, let us find the probability of them purchasing a product under the conditions above.

Here, A = Buy

Applying Bayes Theorem, we get P(A | B) as shown:

$$P(A|B) = P(\text{Yes Buy} \mid \text{Discount} = \text{Yes}, \text{Free Delivery} = \text{Yes}, \text{Day} = \text{Holiday})$$

$$= \frac{P(\text{Discount} = \text{Yes} \mid \text{Yes}) * P(\text{Free Delivery} = \text{Yes} \mid \text{Yes}) * P(\text{Day} = \text{Holiday} \mid \text{Yes}) * P(\text{Yes Buy})}{P(\text{Discount}=\text{Yes}) * P(\text{Free Delivery}=\text{Yes}) * P(\text{Day}=\text{Holiday})}$$

$$= \frac{(19/24) * (21/24) * (8/24) * (24/30)}{(20/30) * (23/30) * (11/30)}$$

$$= 0.986$$

From the two calculations above, we find that:

Probability of purchase = 0.986

Probability of no purchase = 0.178

Finally, we have a conditional probability of purchase on this day.

Next, normalize these probabilities to get the likelihood of the events:

Sum of probabilities = 0.986 + 0.178 = 1.164

Likelihood of purchase = 0.986 / 1.164 = 84.71 percent

Likelihood of no purchase = 0.178 / 1.164 = 15.29 percent

Result: As 84.71 percent is greater than 15.29 percent, we can conclude that an average customer will buy on holiday with a discount and free delivery.

Model parameters :

1. Prior Probabilities: The prior probabilities for each class in the target variable.
2. Smoothing Factor (Laplace smoothing): This is used to avoid division by zero when calculating probabilities for unseen data.
3. Model Type: There are three common types of Naive Bayes models: Gaussian, Multinomial, and Bernoulli. The choice of model depends on the type of data and the problem being solved.
4. Feature Selection: Choosing the most relevant features can improve the accuracy of the model. Feature selection techniques like chi-squared test and mutual information can be used to select the best features.

- **Here is a summary of how Naive Bayes works:**

Naive Bayes is a probabilistic machine learning algorithm that is used for classification problems. It is based on Bayes' theorem, which states that the probability of a class given the features is equal to the probability of the features given the class multiplied by the prior probability of the class divided by the prior probability of the features. In the case of Naive Bayes, the algorithm makes the assumption that the features are independent, given the class. This assumption is known as the "naive" aspect of the algorithm, and it simplifies the calculation of the probability of the features given the class.

There are three main variants of Naive Bayes: Gaussian Naive Bayes, Multinomial Naive Bayes, and Bernoulli Naive Bayes. The choice of the variant depends on the nature of the data and the distribution of the features.

Gaussian Naive Bayes is used when the features are continuous and have a Gaussian distribution. In this case, the probability of the features given the class is modeled as a Gaussian distribution, and the mean and variance of the distribution are estimated from the training data.

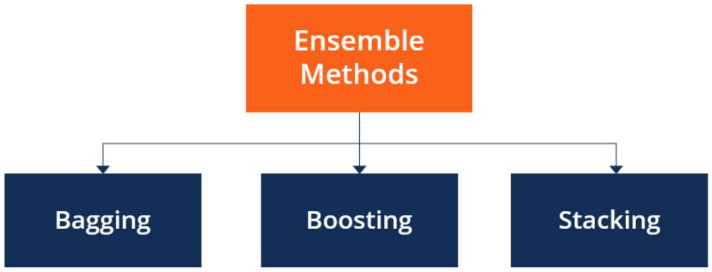
Multinomial Naive Bayes is used when the features are discrete and represent count data, such as the number of occurrences of a word in a text. In this case, the probability of the features given the class is modeled as a multinomial distribution, and the parameters of the distribution are estimated from the training data.

Bernoulli Naive Bayes is used when the features are binary and represent the presence or absence of a feature, such as the presence of a word in a text. In this case, the probability of the features given the class is modeled as a Bernoulli distribution, and the parameters of the distribution are estimated from the training data.

Once the parameters of the distributions are estimated, the algorithm can make predictions for new data points by calculating the probability of each class given the features and choosing the class with the highest probability.

Ensemble methods: bagging, boosting and stacking

Ensemble learning: is a machine learning paradigm where multiple models (often called “weak learners”) are trained to solve the same problem and combined to get better results. The main hypothesis is that when weak models are correctly combined we can obtain more accurate and/or robust models.

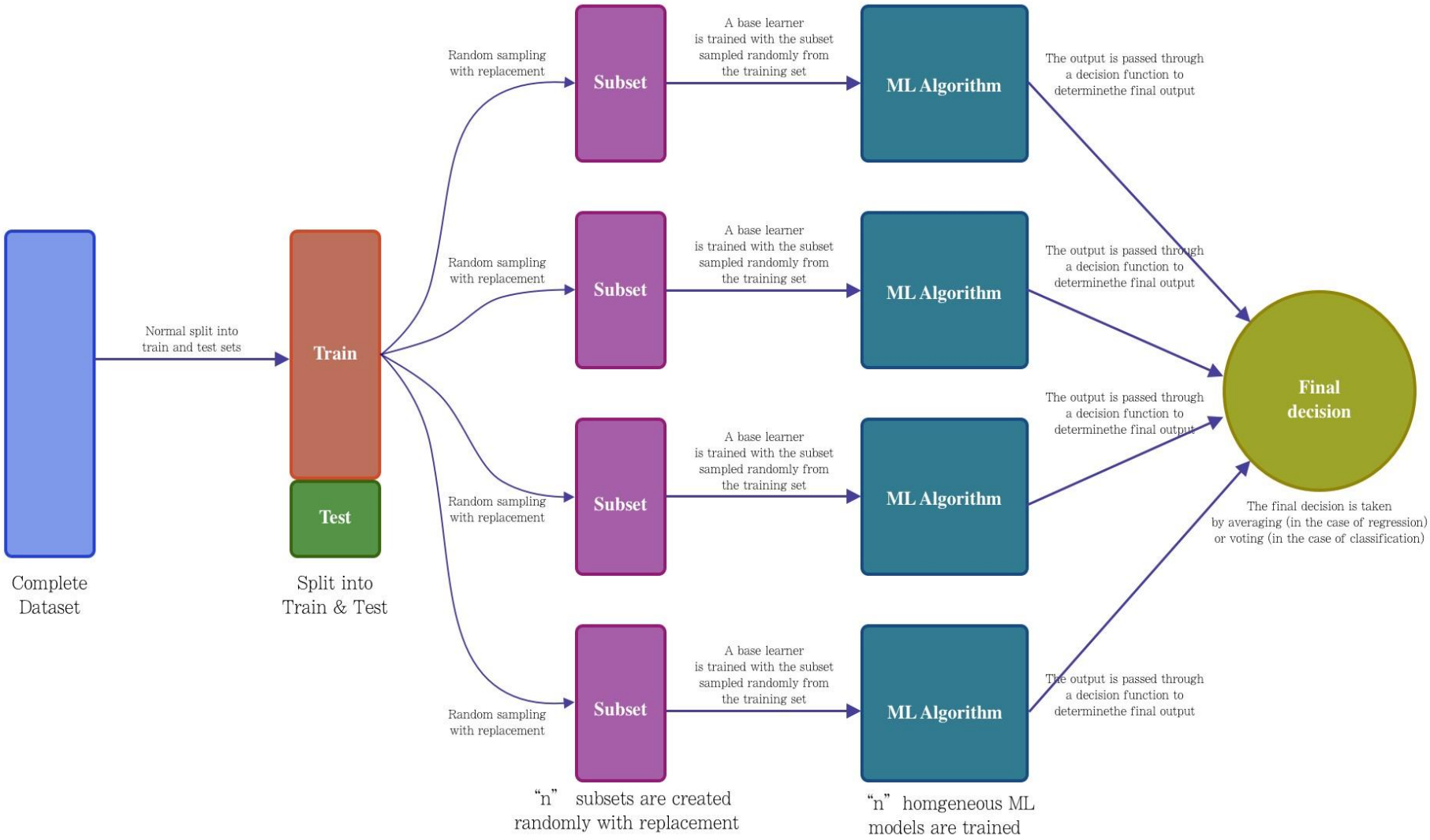


Types Of Ensemble Methods

Ensemble Methods can be used for various reasons, mainly to:

- Decrease Variance *(Bagging)*
- Decrease Bias *(Boosting)*
- Improve Predictions *(Stacking)*

- **Bagging (Bootstrap Aggregating):** is a type of ensemble learning that trains multiple models independently on different random subsets of the training data and aggregates their predictions by majority vote (for classification) or average (for regression). The idea behind bagging is to reduce the variance of the predictions by combining the outputs of multiple models that were trained on different samples of the data. This helps to prevent overfitting, which can occur when a single model is trained on the entire training data.


Complete Dataset
Normal split into train and test sets
Train
Test
Split into Train & Test
Random sampling with replacement
Random sampling with replacement
Random sampling with replacement
Random sampling with replacement
Subset
Subset
Subset
Subset
"n" subsets are created randomly with replacement
A base learner is trained with the subset sampled randomly from the training set
A base learner is trained with the subset sampled randomly from the training set
A base learner is trained with the subset sampled randomly from the training set
A base learner is trained with the subset sampled randomly from the training set
ML Algorithm
ML Algorithm
ML Algorithm
ML Algorithm
"n" homgeneous ML models are trained
The output is passed through a decision function to determinethe final output
The output is passed through a decision function to determinethe final output
The output is passed through a decision function to determinethe final output
The output is passed through a decision function to determinethe final output
Final decision
The final decision is taken by averaging (in the case of regression) or voting (in the case of classification)

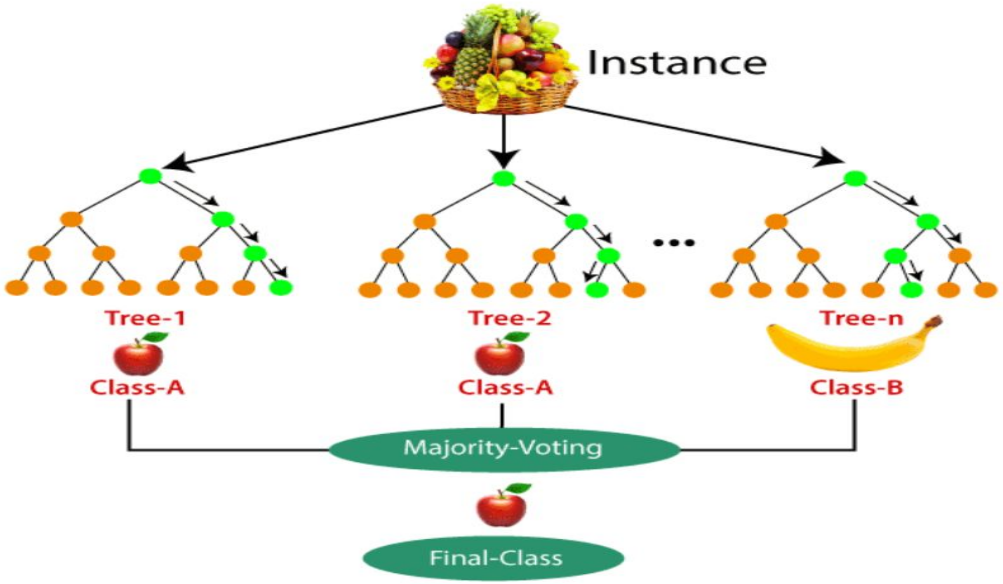
- Some common types of bagging methods are:

1. **Random Forest:** consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction.

- Steps involved in random forest algorithm:

1. In Random forest n number of random records are taken from the data set having k number of records.
2. Individual decision trees are constructed for each sample.
3. Each decision tree will generate an output.
4. Final output is considered based on *Majority Voting or Averaging* for Classification and regression respectively.

For example: consider the fruit basket as the data as shown in the figure below. Now n number of samples are taken from the fruit basket and an individual decision tree is constructed for each sample. Each decision tree will generate an output as shown in the figure. The final output is considered based on majority voting. In the below figure you can see that the majority decision tree gives output as an apple when compared to a banana, so the final output is taken as an apple.



- Important Features of Random Forest:

- **Diversity**: Not all attributes/variables/features are considered while making an individual tree, each tree is different.
- **Parallelization:** Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.
- **Train-Test split:** In a random forest we don't have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.

- Difference Between Decision Tree & Random Forest:

Decision trees	Random Forest
1. Decision trees normally suffer from the problem of overfitting if it's allowed to grow without any control.	1. Random forests are created from subsets of data and the final output is based on average or majority ranking and hence the problem of overfitting is taken care of.
2. A single decision tree is faster in computation.	2. It is comparatively slower.
3. When a data set with features is taken as input by a decision tree it will formulate some set of rules to do prediction.	3. Random forest randomly selects observations, builds a decision tree and the average result is taken. It doesn't use any set of formulas.

- **Model Parameters:**
- **n_estimators**: number of trees the algorithm builds before averaging the predictions.
- **max_features*:*** maximum number of features random forest considers splitting a node.
- **mini_sample_leaf:** determines the minimum number of leaves required to split an internal node.
- **n_jobs:** it tells the engine how many processors it is allowed to use. If the value is 1, it can use only one processor but if the value is -1 there is no limit.
- **Bootstrapping:** The first step in building a Random Forest is to bootstrap the training data, which means that multiple samples of the data are drawn with replacement. This creates different versions of the training data, which are used to train multiple decision trees.
- **criterion:** the function used to measure the quality of a split. The default value is "gini" for classification problems, and "mse" for regression problems.

Impurity	Task	Formula	Description
Gini impurity	Classification	$\sum_{i=1}^{C} f_i(1-f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels.
Entropy	Classification	$\sum_{i=1}^{C} -f_i \log(f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels.
Variance / Mean Square Error (MSE)	Regression	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\mu)^2$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{N} y_i$
Variance / Mean Absolute Error (MAE) (Scikit-learn only)	Regression	$\frac{1}{N}\sum_{i=1}^{N}\lvert y_i-\mu\rvert$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{N} y_i$

- **Here is a summary of how Random forest works:**

Random Forest is an ensemble learning algorithm used for both regression and classification tasks. It is based on the idea of combining multiple decision trees to form a forest of trees, where each tree represents a single decision, and the forest represents a collective decision. The final prediction of the Random Forest algorithm is obtained by combining the predictions of all the individual trees in the forest.

A decision tree is a tree-like model that makes decisions based on a series of simple rules. Each node in the tree represents a test on a feature, and the branches represent the possible outcomes of the test. The leaves of the tree represent the final prediction of the tree, which can be a class label in the case of classification or a continuous value in the case of regression.

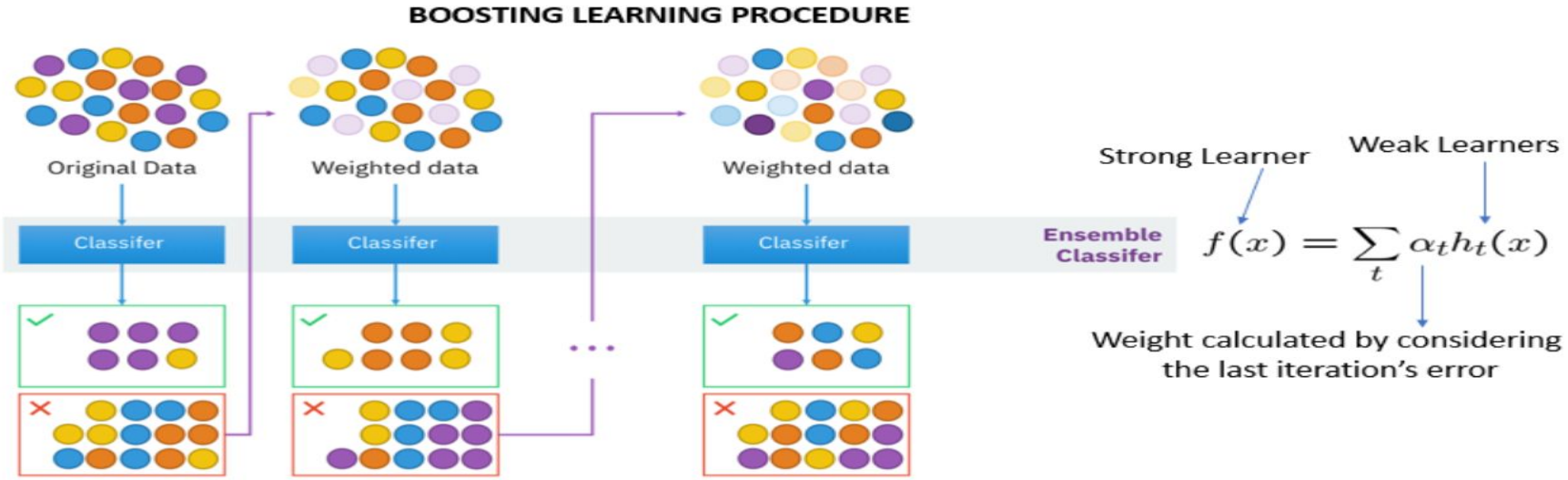
In Random Forest, multiple decision trees are grown in parallel, each from a random subset of the data and a random subset of the features. This means that each tree will be different from the others, and the combination of all the trees will form a diverse set of rules that can handle different aspects of the data. By combining the predictions of all the trees, the Random Forest algorithm can obtain a more accurate and robust prediction compared to a single decision tree.

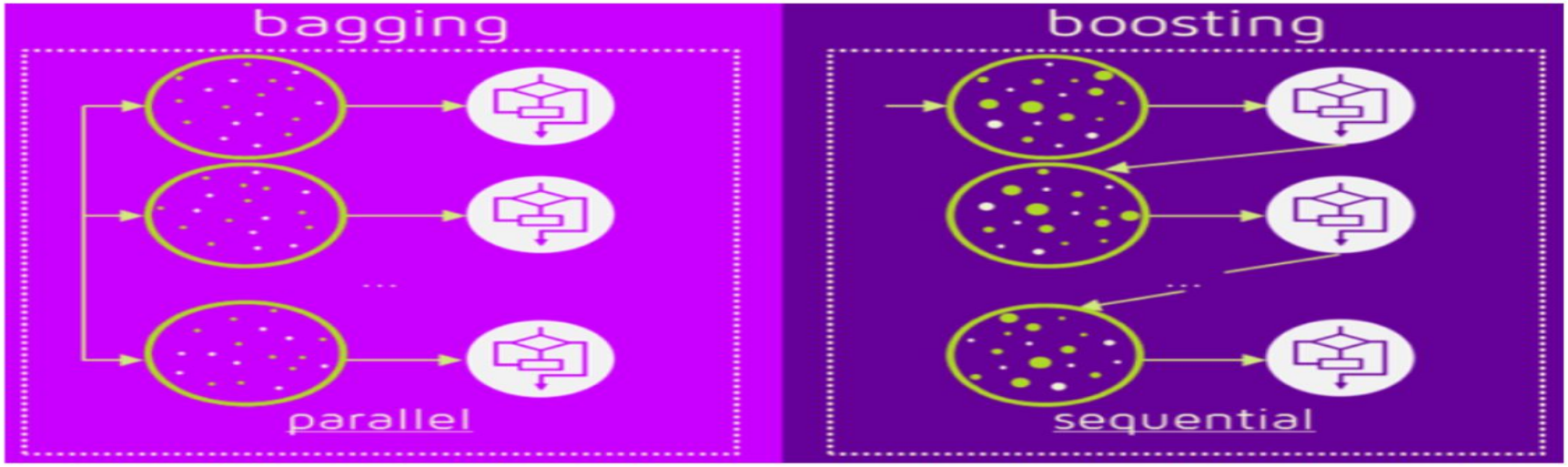
When growing a decision tree in Random Forest, the algorithm uses a technique called bootstrapping, where each tree is grown from a random sample of the data with replacement. This means that some data points may appear multiple times in the sample, while others may not appear at all. This creates a diverse set of training data for each tree, which helps to reduce overfitting and increase the generalization of the model.

In addition to bootstrapping, Random Forest also uses a technique called feature bagging, where each tree is grown from a random subset of the features. This means that each tree will consider a different set of features, and the combination of all the trees will form a diverse set of tests on different combinations of features. This helps to reduce the correlation between the trees and increase the diversity of the rules used to make predictions.

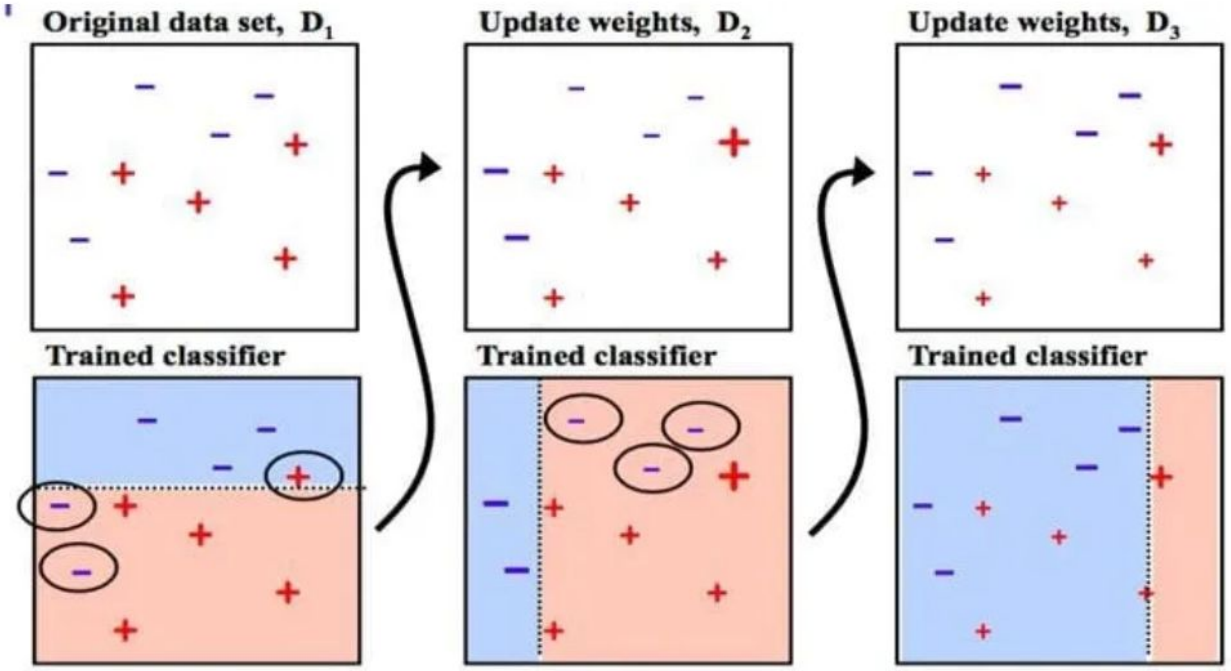
Once the trees are grown, the Random Forest algorithm can be used to make predictions for new data points by passing the data point down each tree and aggregating the predictions of all the trees. The aggregation can be done by taking the majority vote in the case of classification, or by taking the average in the case of regression.

- **Boosting:** is a type of ensemble learning that trains multiple models sequentially, each trying to correct the mistakes of the previous model. The final prediction is made by combining the predictions of all models. Boosting algorithms work by giving more weight to examples that are misclassified by previous models, so that the subsequent models can focus more on correcting these mistakes.





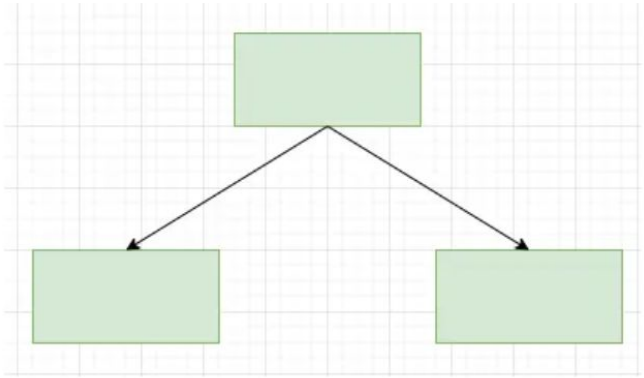
- **AdaBoost (Adaptive Boosting):** is a supervised machine learning algorithm. This algorithm can solve both kinds of problem statements that is classification and regression. It comes under the category of boosting ensemble technique. The basic idea of this algorithm is to collect some weak machine learning algorithms and train them in a sequential order where each weak learner passes some kind of helpful information to the next weak learner so that the next weak learner can learn better. This is why it is called Adaptive boosting which is the full form of Adaboost. The last step is to aggregate the outputs of the trained learner using some mathematical function.



Adaboost Algorithm

- **How do Adaboost works?**
- **Prerequisites of an Adaboost:**

  - **Weak Learners:** Those machine learning models give very low accuracy. The accuracy range must be from 50% to 60%.
  - **Decision Stump:**They are just a normal decision tree with depth or height equal to one. Adaboost only uses decision stumps as its base estimator.



Decision Stump

  - **Up-Sampling:** This is a technique used in adaboost to update the dataset by creating new weights for the rows. The new weights are generated by using the below formula:

$$new\ weight = current\ weight \times e^{\pm\alpha}$$

    - The sum of new weights will not be equal to 1 so, we will perform normalization.

- **Alpha:** Alpha is the weight for the respective decision stump. This decides how much the decision stump will affect the final result. If the model is good then the value of alpha will increase.

1. Add a column to the dataset which contains the weight of every row. The initial weight is calculated by assigning equal weights to all the rows by using the below formula:

$$weight = \frac{1}{n}$$

Where n = number of rows.
NOTE : If we add all the weight we will get 1.

2. Train a decision stump on the dataset.
3. Make predictions using the trained decision stump.
4. Calculate alpha for the decision stump by using the below formula:

$$alpha(\alpha) = \frac{1}{2}\ln\left(\frac{1 - error}{error}\right)$$

Where error = sum of weights of rows whose prediction is wrong.

5. Increase the importance of the miss predicted values rows by using the technique of upsampling and updating the dataset.

6. We will repeat all the steps again on a new decision stump but by using the updated dataset.
7. Now the last step, after running for n estimators we will get:

$$\alpha = \alpha_1 + \alpha_2 + \alpha_3 + \ldots\ldots \alpha_n$$

And

$$m = m_1 + m_2 + m_3 + \ldots\ldots m_n$$

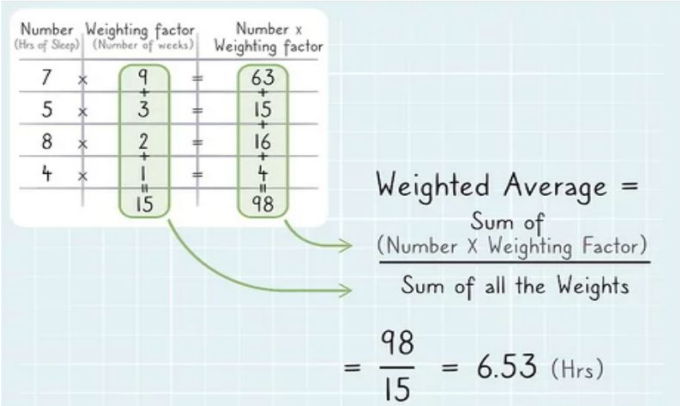
Where m = decision stump

$$Output = \alpha_1 m_1 + \alpha_2 m_2 + \alpha_3 m_3 + \ldots\ldots \alpha_n m_n$$

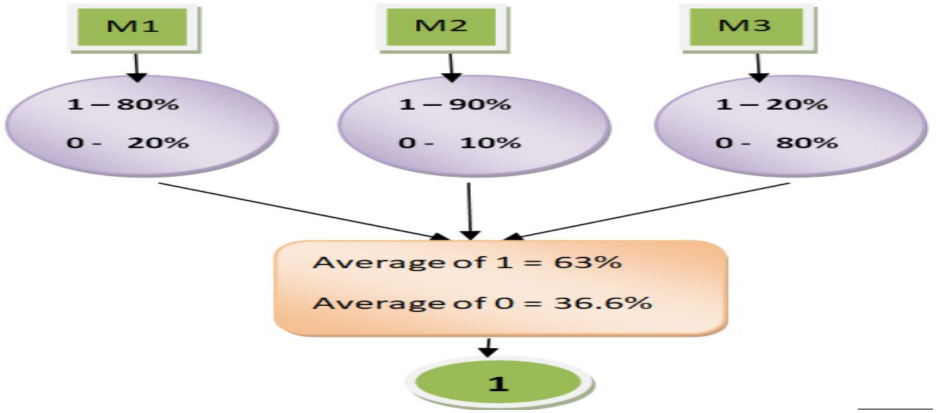
Pass this "output" in any function ranging from 0 to 1

$$Final\ Output = \sin(Output)$$

- The final classifier is formed by combining the outputs of the weak learners using a weighted majority vote, where the weight of each weak learner depends on its accuracy.
- The final regressor is formed by combining the predictions of the weak learners using a weighted average, where the weight of each weak learner depends on its prediction error.



weighted average



weighted majority vote

- **model parameter:**

1. **Number of weak learners (T):** This parameter controls the number of weak learners that are combined to form the final strong classifier or regressor. Larger values of T typically lead to better performance, but also result in a slower model.
2. **Weak learner algorithm:** This parameter determines the type of weak learner used in the AdaBoost algorithm. Common choices include decision trees, support vector machines, and neural networks. The choice of weak learner can have a significant impact on the performance of the final model.
3. **Learning rate (eta):** This parameter controls the relative weight assigned to each weak learner in the final classifier or regressor. Larger values of eta result in more weight being assigned to each weak learner, leading to a stronger model.
4. **Sampling strategy:** This parameter determines the way in which samples are selected to train each weak learner. Common choices include random sampling, stratified sampling, and boosting-based sampling.
5. **Regularization parameters:** Some weak learner algorithms, such as support vector machines and neural networks, have their own regularization parameters that can be adjusted to control their behavior. These parameters can impact the performance of the final AdaBoost model.

- **Here is a summary of how Adaboost works:**

AdaBoost (Adaptive Boosting) is a type of ensemble learning algorithm used for classification problems. It combines multiple weak learners to form a strong learner that can accurately classify the data. A weak learner is a simple model that performs better than random guessing, but not necessarily as well as a strong model.

The basic idea behind AdaBoost is to iteratively improve the performance of the model by giving more weight to the samples that were misclassified in the previous iteration. In each iteration, a new weak learner is trained on the weighted data, where the samples that were misclassified in the previous iteration are given more weight. The predictions of the new weak learner are then combined with the predictions of the previous weak learners to form the final prediction.

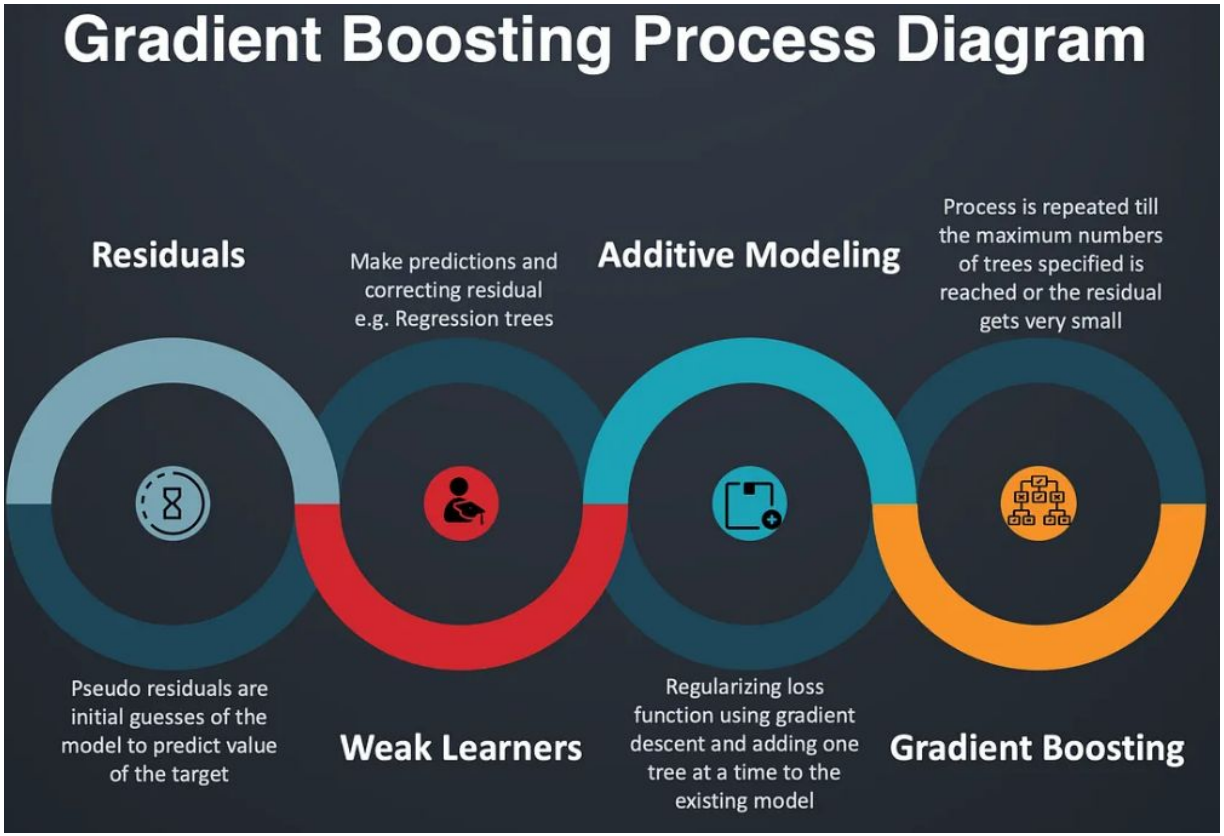
The combination of the weak learners can be done by assigning a weight to each weak learner, which represents the importance of the weak learner in the final prediction. The weights are updated in each iteration based on the accuracy of the weak learner. The weak learners that perform better are given more weight, while the weak learners that perform poorly are given less weight.

The final prediction of AdaBoost is obtained by taking a weighted majority vote of the predictions of all the weak learners. The prediction of each weak learner is assigned a weight based on the accuracy of the weak learner, and the prediction with the highest weight is selected as the final prediction.

AdaBoost can be used with any type of weak learner, but decision trees are a common choice because they are simple to implement and can handle complex decision boundaries.

- **Gradient boosting:**

The main idea behind this algorithm is to build models sequentially and these subsequent models try to reduce the errors of the previous model.
When the target column is continuous, we use **Gradient Boosting Regressor** whereas when it is a classification problem, we use **Gradient Boosting Classifier.** The only difference between the two is the *"Loss function"*. The objective here is to minimize this loss function by adding weak learners using gradient descent. Since it is based on loss function hence for regression problems, we'll have different loss functions like Mean squared error (MSE) and for classification, we will have different for e.g log-likelihood.



- **Gradient Boosting Regressor:**

Following is a sample from a random dataset where we have to predict the car price based on various features. The target column is price and other features are independent features.

Row No.	Cylinder Number	Car Height	Engine Location	Price
1	Four	48.8	Front	12000
2	Six	48.8	Back	16500
3	Five	52.4	Back	15500
4	Four	54.3	Front	14000

Step 1: The first step in gradient boosting is to build a base model to predict the observations in the training dataset. We take an average of the target column and assume that to be the predicted value as shown below:

Row No.	Cylinder Number	Car Height	Engine Location	Price	Prediction 1
1	Four	48.8	Front	12000	14500
2	Six	48.8	Back	16500	14500
3	Five	52.4	Back	15500	14500
4	Four	54.3	Front	14000	14500

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

- L: Loss function.
- gamma: predicted value.
- argmin means: to find a predicted value/gamma for which the loss function is minimum.

Since the target column is continuous our loss function will be:

$$L = \frac{1}{n} \sum_{i=0}^{n} \left(y_i - \gamma_i \right)^2$$

- yi: observed value.
- gamma: predicted value.

Now we need to find a minimum value of gamma such that this loss function is minimum.

$$\frac{\mathrm{dL}}{\mathrm{d}\gamma} = \frac{2}{2} \left(\sum_{i=0}^{n} \left(y_i - \gamma_i \right) \right) = - \sum_{i=0}^{n} \left(y_i - \gamma_i \right)$$

$$L = \frac{1}{2}(12000-\gamma)^2 + \frac{1}{2}(16500-\gamma)^2 + \frac{1}{2}(15500-\gamma)^2 + \frac{1}{2}(14000-\gamma)^2$$

$$\frac{dL}{d\gamma} = \frac{2}{2}(12000-\gamma)(-1) + \frac{2}{2}(16500-\gamma)(-1) + \frac{2}{2}(15500-\gamma)(-1) + \frac{2}{2}(14000-\gamma)(-1)$$

$$Now\ \frac{dL}{d\gamma} = 0\ and\ taking\ (-)\ common$$

$$\Rightarrow -[12000-\gamma+16500-\gamma+15500-\gamma+14000-\gamma] = 0$$

$$\Rightarrow [58000 - 4\gamma] = 0$$

$$\Rightarrow 58000 = 4\gamma$$

$$\Rightarrow \gamma = \frac{58000}{4} = 14500$$

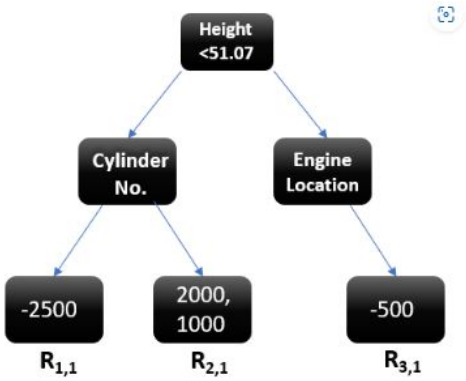
- **Step 2:** The next step is to calculate the pseudo residuals which are (observed value – predicted value).

Row No.	Cylinder Number	Car Height	Engine Location	Price	Prediction 1	Residual 1
1	Four	48.8	Front	12000	14500	-2500
2	Six	48.8	Back	16500	14500	2000
3	Five	52.4	Back	15500	14500	1000
4	Four	54.3	Front	14000	14500	-500

- **Step 3:** we will build a model on these pseudo residuals and make predictions. Because we want to minimize these residuals and minimizing the residuals will eventually improve our model accuracy and prediction power.

Step 4: In this step we find the output values for each leaf of our decision tree. That means there might be a case where 1 leaf gets more than 1 residual, hence we need to find the final output of all the leaves. To find the output we can simply take the average of all the numbers in a leaf, doesn't matter if there is only 1 number or more than 1.

- Let's understand this even better with the help of an example. Suppose this is our regressor tree:



- We see 1st residual goes in $R_{1,1}$,2nd and 3rd residuals go in $R_{2,1}$ and 4th residual goes in $R_{3,1}$.

- Let's calculate the output for the first leave that is $R_{1,1}$

$$\gamma_{1,1} = argmin \frac{1}{2}\Big(12000 - (14500 + \gamma)\Big)^2$$

$$\gamma_{1,1} = argmin \frac{1}{2}\Big(-2500 - \gamma\Big)^2$$

Now we need to find the value for gamma for which this function is minimum. So we find the derivative of this equation w.r.t gamma and put it equal to 0.

$$\frac{d}{d\gamma}\frac{1}{2}\Big(-2500 - \gamma\Big)^2 = 0$$

$$-2500 - \gamma = 0$$

$$\gamma = -2500$$

Hence the leaf $R_{1,1}$ has an output value of -2500. Now let's solve for the $R_{2,1}$

$$\gamma_{2,1} = \text{argmin}\left[\frac{1}{2}\Big(16500-(14500+\gamma)\Big)^2+\frac{1}{2}\Big(15500-(14500+\gamma)\Big)^2\right]$$

$$\gamma_{2,1} = \text{argmin}[\frac{1}{2}(2000-\gamma)^2+\frac{1}{2}(1000-\gamma)^2]$$

Let's take the derivative to get the minimum value of gamma for which this function is minimum:

$$\frac{d}{d\gamma}\left[\frac{1}{2}(2000-\gamma)^2+\frac{1}{2}\Big(1000-\gamma\Big)^2\right] = 0$$

$$2000-\gamma+1000-\gamma=0$$

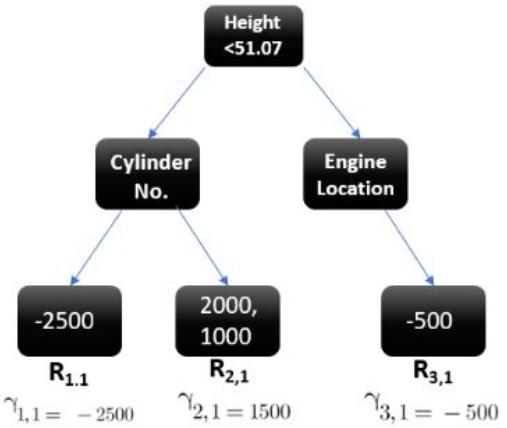
$$3000-2\gamma=0$$

$$\frac{3000}{2}=\gamma$$

$$\gamma=1500$$

We end up with the average of the residuals in the leaf R2,1 . Hence if we get any leaf with more than 1 residual, we can simply find the average of that leaf and that will be our final output.

- Now after calculating the output of all the leaves, we get:



Step:5 This is finally the last step where we have to update the predictions of the previous model. It can be updated as:

$$\text{Update the model:}$$
$$F_m(x) = F_{m-1}(x) + \nu_m h_m(x)$$

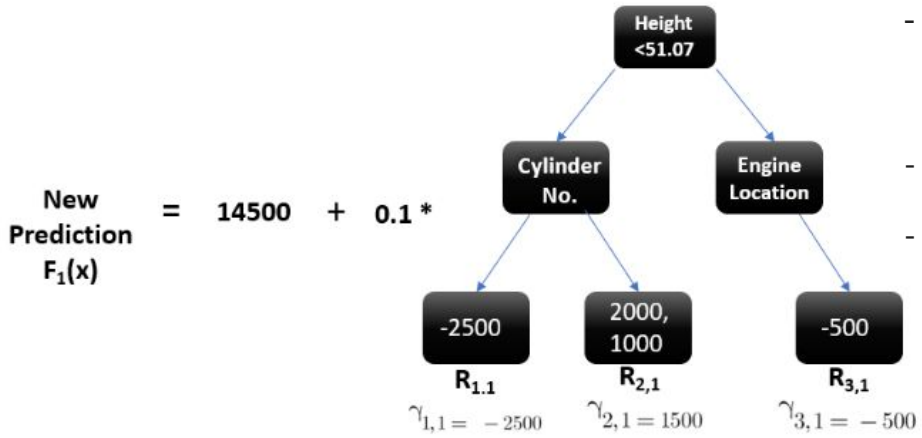
- where m is the number of decision trees made.
- Since we have just started building our model so our m=1. Now to make a new DT our new predictions will be:

Fm-1(x): the prediction of the base model (previous prediction) since F1-1=0 , F0 is our base model hence the previous prediction is 14500.

nu(v) : the *learning rate* that is usually selected between 0-1. It reduces the effect each tree has on the final prediction, and this improves accuracy in the long run. Let's take *nu=0.1* in this example.

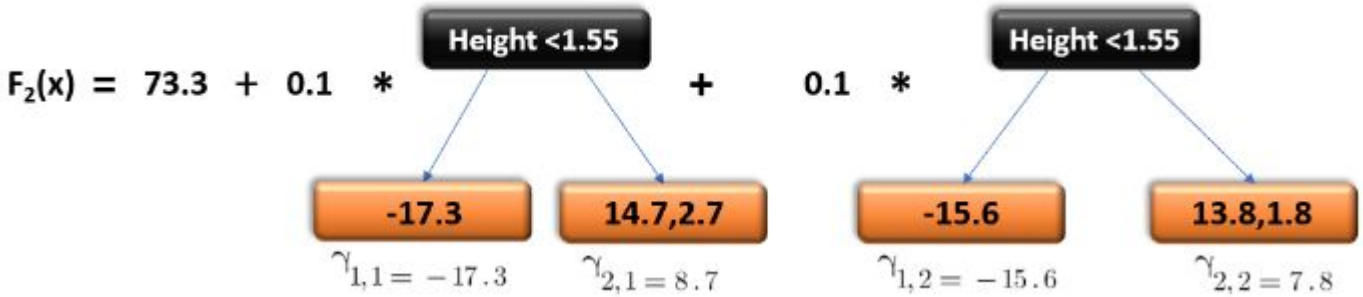
Hm(x): the recent DT made on the residuals.

Let's calculate the new prediction now:



- Suppose we want to find a prediction of our first data point which has a car height of 48.8. This data point will go through this decision tree and the output it gets will be multiplied with the learning rate and then added to the previous prediction.
- let's say m=2 which means we have built 2 decision trees and now we want to have new predictions.
- This time we will add the previous prediction that is F1(x) to the new DT made on residuals. We will iterate through these steps again and again till the *loss is negligible.*

below is just a hypothetical example to understand how this predicts for a new dataset:



If a new data point says height = 1.40 comes, it'll go through all the trees and then will give the prediction. Here we have only 2 trees hence the datapoint will go through these 2 trees and the final output will be $F_2(x)$.

- **Gradient Boosting Classifier:** A gradient boosting classifier is used when the target column is binary. All the steps explained in the Gradient boosting regressor are used here, the only difference is we change the loss function.

$$L = -\sum_{i=1}^{n} y_i \log(p) + (1-p)\log(1-p)$$

  - to calculate the output of a leaf:

$$\gamma = \frac{\sum_{i=1}^{n} \text{Residual}_i}{\sum_{i=1}^{n} [\text{Previous probability}_i \times (1 - \text{Previous probability}_i)]}$$

we are ready to get new predictions by adding our base model with the new tree we made on residuals.

- **Here is a summary of how gradient boosting works:**

1. **Initial Model:** The first step is to fit a simple model to the data, such as a decision tree. This model is referred to as the base model.
2. **Residuals:** Next, the residuals (the differences between the predicted values and the actual values) of the base model are calculated.
3. **New Model:** A new model is then fit to the residuals of the previous model, with the goal of reducing the residuals as much as possible. The new model is added to the base model to produce a combined model.
4. **Iteration:** This process is repeated multiple times, each time fitting a new model to the residuals of the previous iteration. The number of iterations is determined by the user and is a trade-off between accuracy and computational efficiency.
5. **Final Model:** After a specified number of iterations, the final model is the weighted sum of all the individual models. The weights are determined based on the optimization of a loss function, such as mean squared error for regression problems and cross-entropy for classification problems.

- **Model parameters:**
- **n_estimators:** This parameter determines the number of individual trees that are fit to the residuals in the boosting process. A larger number of trees results in a more complex model, but also increases the risk of overfitting.
- **Learning Rate:** determines the size of the contribution of each individual tree to the final model. A smaller learning rate means that each tree has a smaller impact on the final model, which reduces the risk of overfitting but also requires more boosting iterations to reach the optimal solution.
- **max_depth:** This parameter determines the maximum depth of the individual trees in the model. A smaller maximum depth results in simpler trees and reduces the risk of overfitting, while a larger maximum depth results in more complex trees and a better fit to the data.
- **subsample**: This parameter determines the fraction of the training samples that are used in each iteration.

- **XGBoost (Extreme Gradient Boosting):** XGBoost stands for Extreme Gradient Boosting. It's a parallelized and carefully optimized version of the gradient boosting algorithm. Parallelizing the whole boosting process hugely improves the training time.

  - **Algorithm Enhancements:**
    1. Tree Pruning : Pruning is a machine learning technique to reduce the size of regression trees by replacing nodes that don't contribute to improving classification on leaves. The idea of pruning a regression tree is to prevent overfitting of the training data. The most efficient method to do pruning is Cost Complexity or Weakest Link Pruning which internally uses mean square error, k-fold cross-validation and learning rate. XGBoost creates nodes (also called splits) up to *max_depth* specified and starts pruning from backward until the loss is below a threshold. Consider a split that has -3 loss and the subsequent node has +7 loss, XGBoost will not remove the split just by looking at one of the negative loss. It will compute the total loss (-3 + 7 = +4) and if it turns out to be positive it keeps both.
    2. Sparsity Aware Split Finding : It is quite common that the data we gather has sparsity (a lot of missing or empty values) or becomes sparse after performing data engineering (feature encoding). To be aware of the sparsity patterns in the data, a default direction is assigned to each tree. XGBoost handles missing data by assigning them to default direction and finding the best imputation value so that it minimizes the training loss. Optimization here is to visit only missing values which make the algorithm run 50x faster than the naïve method.

- **System Enhancements:**

1. **Parallelization**: Tree learning needs data in a sorted manner. To cut down the sorting costs, data is divided into compressed blocks (each column with corresponding feature value). XGBoost sorts each block parallelly using all available cores/threads of CPU. This optimization is valuable since a large number of nodes gets created frequently in a tree. In summary, XGBoost parallelizes the sequential process of generating trees.
2. **Cache Aware**: By cache-aware optimization, we store gradient statistics (direction and value) for each split node in an internal buffer of each thread and perform accumulation in a mini-batch manner. This helps to reduce the time overhead of immediate read/write operations and also prevent cache miss. Cache awareness is achieved by choosing the optimal size of the block (generally 2^{16}).

- **Flexibility in XGBoost:**

1. **Customized Objective Function** : An objective function intends to maximize or minimize something. In ML, we try to minimize the objective function which is a combination of the loss function and regularization term.

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

Loss function ↙ ↘ Regularization

- *L(Φ)*: objective function.

- Optimizing the loss function encourages predictive models whereas optimizing regularization leads to smaller variance and makes prediction stable. Different objective functions available in XGBoost are:
  - ***reg****: linear* for regression
  - ***reg****: logistic,* and *binary: logistic* for binary classification
  - ***multi****: softmax,* and *multi: softprob* for multiclass classification

- **Customized Evaluation Metric:** This is a metric used to monitor the model's accuracy on validation data.
  - ***rmse****:* Root mean squared error (Regression).
  - ***mae****:* Mean absolute error (Regression).
  - ***error***: Binary classification error (Classification).
  - ***logloss***: Negative log-likelihood (Classification).
  - ***auc***: Area under the curve (Classification).

- **Cross-validation:**

Built-in Cross-validation: Cross validation is a statistical method to evaluate machine learning models on unseen data. It comes in handy when the dataset is limited and prevents overfitting by not taking an independent sample (holdout) from training data for validation. By reducing the size of training data, we are compromising with the features and patterns hidden in the data which can further induce errors in our model. This is similar to *cross_val_score* functionality provided by the scikit-learn library.

- *XGBoost uses built-in cross validation function cv().*

- **k-fold Cross-validation:** In k-fold cross-validation, data is shuffled and divided into *k* equal sized subsamples. One of the *k* subsamples is used as a test/validation set and remaining (*k -1*) subsamples are put together to be used as training data. Then we fit a model using training data and evaluate it using the test set. This process is repeated *k* times so that every data point stays in validation set exactly once. The *k* results from each model should be averaged to get the final estimation. The advantage of this method is that we significantly reduce bias, variance, and also increase the robustness of the model.

    - Model tuning in XGBoost can be implemented by cross-validation strategies like GridSearchCV and RandomizedSearchCV.

        1. **Grid Search**: We pass on a parameter's dictionary to the function and compare the cross-validation score for each combination of parameters (many to many) in the dictionary and return the set having the best parameters.
        2. **Random Search**: We draw a random value during each iteration from the range of specified values for each hyperparameter searched over, and evaluate a model with those hyperparameters. After completing all iterations, it picks the hyperparameter configuration with the best score.

    - **Plot Importance Module:** XGBoost library provides a built-in function to plot features ordered by their importance. The function is *plot_importance(model)* and it takes the trained model as its parameter. The function gives an informative bar chart representing the significance of each feature and names them according to their index in the dataset. The importance is calculated based on an *importance_type* variable which takes the parameters:
    - **weights** (default):tells the times a feature appears in a tree
    - **gain:**is the average training loss gained when using a feature
    - **cover:** which tells the coverage of splits, i.e. the number of times a feature is used weighted by the total training point that falls in that branch.

- **Here is a summary of how XGBoost works:**

XGBoost (Extreme Gradient Boosting) is a type of gradient boosting algorithm used for both regression and classification tasks. It is a powerful and efficient algorithm that has become one of the most widely used machine learning algorithms in industry and academia.

Gradient Boosting is an ensemble learning algorithm that combines multiple weak models to form a strong model. In each iteration, a new model is trained to correct the mistakes made by the previous models. The final prediction of the Gradient Boosting algorithm is obtained by combining the predictions of all the models.

XGBoost improves upon the traditional Gradient Boosting algorithm by using a more efficient optimization algorithm, which is based on gradient descent. The optimization algorithm is used to find the optimal values of the model parameters that minimize the loss function. The loss function measures the difference between the predicted values and the actual values, and the goal of the optimization algorithm is to minimize this difference.

In XGBoost, the weak models are decision trees, and the decision trees are grown in a greedy manner, where each tree tries to correct the mistakes made by the previous trees. The prediction of each tree is added to the final prediction of the model, and the prediction is updated in each iteration.

XGBoost also includes several techniques to improve the performance and efficiency of the algorithm, such as regularization, feature selection, and early stopping. Regularization is used to prevent overfitting by adding a penalty term to the loss function that discourages the model from having too many parameters. Feature selection is used to select the most important features to include in the model, which helps to reduce the dimensionality of the data and improve the performance of the algorithm. Early stopping is used to stop the training of the model when the performance on the validation data stops improving, which helps to prevent overfitting and improve the generalization of the model.

In summary, XGBoost (Extreme Gradient Boosting) is a type of gradient boosting algorithm used for both regression and classification tasks. It is a powerful and efficient algorithm that has become one of the most widely used machine learning algorithms in industry and academia. XGBoost improves upon the traditional Gradient Boosting algorithm by using a more efficient optimization algorithm, which is based on gradient descent. XGBoost also includes several techniques to improve the performance and efficiency of the algorithm, such as regularization, feature selection, and early stopping.

- **Model parameters**:
- **Booster:** This parameter specifies the type of booster to use, which is either "gbtree" (gradient boosting decision tree) or "gblinear" (gradient boosting linear model).
- **eta (learning rate):** This parameter controls the learning rate for the optimization algorithm and determines how quickly the model updates its parameters.
- **max_depth:** This parameter controls the maximum depth of the decision trees used in the model. Larger values of max_depth result in deeper trees and more complex models.
- **subsample:** This parameter controls the fraction of the training data used in each iteration of the optimization algorithm.
- **objective:** This parameter specifies the type of problem being solved, such as binary classification, multiclass classification, or regression.
- **n_estimators:** This parameter controls the number of decision trees used in the model.

- **CatBoost:** supports numerical, categorical, and text features but has a good handling technique for categorical data.

  - The CatBoost algorithm has quite a number of parameters to tune the features in the processing stage.

    - **Features of CatBoost:**

      - **Robust:** can improve the performance of the model while reducing overfitting and the time spent on tuning.
      - **Accuracy:** is a high performance and greedy novel gradient boosting implementation.
      - **Categorical Features Support:** The key features of CatBoost is one of the significant reasons why it was selected by many boosting algorithms such as LightGBM, XGBoost algorithm. With other machine learning algorithms. After preprocessing and cleaning your data, the data has to be converted into numerical features so that the machine can understand and make predictions. This is same like, for any text related models we convert the text data into to numerical data it is know as word embedding techniques. This process of encoding or conversion is time-consuming. CatBoost supports working with non-numeric factors, and this saves some time plus improves your training results.

- **Faster Training & Predictions:** the maximum number of GPUs per server is 8 GPUs. Some data sets are more extensive than that, but CatBoost uses distributed GPUs. This feature enables CatBoost to learn faster and make predictions 13-16 times faster than other algorithms.

- **Here is a summary of how XGBoost works:**

CatBoost is a gradient boosting framework that is designed to work well with categorical features. Here are the key details about how it works:

1. **Gradient boosting:** CatBoost uses gradient boosting to iteratively improve the performance of an ensemble of decision trees. Each new tree is trained to correct the errors made by the previous trees.
2. **Categorical features:** CatBoost is designed to handle categorical features in a more efficient way than other gradient boosting frameworks. It uses an algorithm called Ordered Boosting to process categorical features, which assigns numerical values to each category based on the target variable.
3. **Regularization:** CatBoost uses various forms of regularization to prevent overfitting. This includes L2 regularization, feature importance regularization, and permutation-based feature importance.
4. **Scalability:** CatBoost is designed to be highly scalable, with support for parallel processing and out-of-core learning. This makes it possible to train models on large datasets without running out of memory.
5. **Hyperparameter tuning:** CatBoost provides a wide range of hyperparameters that can be tuned to optimize model performance. These include parameters that control the complexity of the model, the learning rate, and the amount of regularization.
6. **Ordered Boosting:** CatBoost uses an algorithm called Ordered Boosting to process categorical features. Ordered Boosting assigns numerical values to each category based on the target variable and their order in the training data. This means that the numerical values assigned to each category are optimized to minimize the loss function during training.
7. **Feature importance:** CatBoost provides several ways to measure the importance of features in the model. One of these is feature importance regularization, which penalizes the model for relying too heavily on certain features. Another is permutation-based feature importance, which measures the decrease in performance when a feature is randomly shuffled.
8. **Handling missing values:** CatBoost can handle missing values in the input data by using a combination of gradient-based methods and probabilistic inference. This allows the model to make accurate predictions even when some of the input data is missing.
9. **Model architecture:** CatBoost uses a combination of decision trees and boosting to construct the final model. The decision trees are trained using the gradient boosting algorithm, and are combined into an ensemble to improve the accuracy and generalization of the model.

Ordered Boosting: is a variant of gradient boosting that is specifically designed for problems where the target variable has an inherent ordering or ranking. In such problems, the goal is to predict the relative order or ranking of the instances, rather than their absolute values.

The idea behind Ordered Boosting is to train a set of weak models, each of which predicts the relative order of the instances based on a different set of features. The predictions of these weak models are then combined using gradient boosting, in a similar manner as standard gradient boosting.

The key difference between Ordered Boosting and standard gradient boosting is in the loss function used to train the weak models. In Ordered Boosting, the loss function is based on the pairwise ranking of the instances, rather than the absolute difference between the predicted and actual values. Specifically, the loss function penalizes the weak model when it predicts a pairwise ranking that is inconsistent with the actual pairwise ranking of the instances.

In each iteration of Ordered Boosting, a new weak model is trained on a different set of features, using the gradient of the pairwise ranking loss function. The weak model is then added to the ensemble, and the prediction of the ensemble is updated by adding the contribution of the new weak model, multiplied by a learning rate.

The final prediction of the Ordered Boosting ensemble is obtained by summing the predictions of all the weak models, weighted by their contribution to the ensemble.

In summary, Ordered Boosting is a variant of gradient boosting that is designed for problems where the target variable has an inherent ordering or ranking. It trains a set of weak models that predict the pairwise ranking of the instances, and combines their predictions using gradient boosting.

- Model parameters:

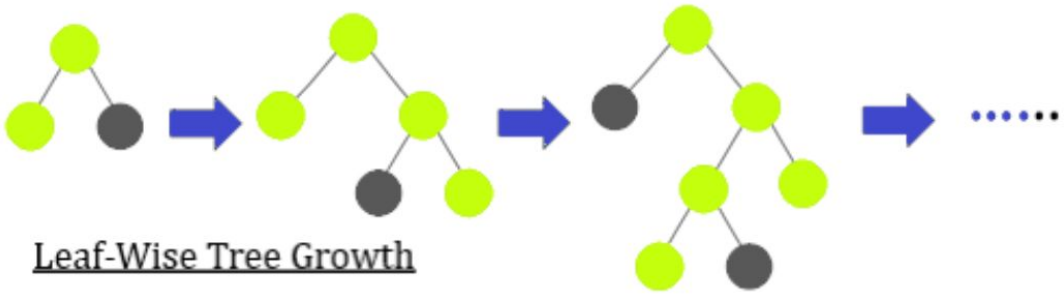
- `learning_rate`: This parameter controls the step size of the gradient descent algorithm used to train the model. A smaller learning rate will result in slower convergence, but may help prevent overfitting.
- `depth`: This parameter controls the maximum depth of the decision trees used in the ensemble. A larger depth can result in a more complex model, but may also lead to overfitting.
- `l2_leaf_reg`: This parameter controls the L2 regularization applied to the weights of the decision trees. Regularization helps prevent overfitting by penalizing large weights.
- `n_estimators`: This parameter controls the number of decision trees in the ensemble. A larger number of trees can result in a more accurate model, but may also lead to overfitting and slower training times.
- `colsample_bylevel`: This parameter controls the fraction of features used to train each level of the decision trees. A smaller value can help prevent overfitting by reducing the correlation between the trees.
- `subsample`: This parameter controls the fraction of instances used to train each tree. A smaller value can help prevent overfitting by reducing the correlation between the trees.
- `cat_features`: This parameter is used to specify which features are categorical. CatBoost can automatically handle categorical features, but this parameter can be used to override the automatic detection or to specify the ordering of the categories.
- `one_hot_max_size`: This parameter controls the maximum number of categories to convert to one-hot encoding. This can be used to prevent the creation of too many new features, which can lead to overfitting.

- **LightGBM (Light Gradient Boosting Machine):** is a gradient boosting framework based on decision trees to increases the efficiency of the model and reduces memory usage. It uses two novel techniques: **Gradient-based One Side Sampling** and **Exclusive Feature Bundling (EFB)** which fulfills the limitations of histogram-based algorithm that is primarily used in all GBDT (Gradient Boosting Decision Tree) frameworks. The two techniques of GOSS and EFB described below form the characteristics of LightGBM Algorithm. They comprise together to make the model work efficiently and provide it a cutting edge over other GBDT frameworks

    - **Gradient-based One Side Sampling Technique:** is a technique used in the LightGBM machine learning algorithm to speed up the training process and improve the accuracy of the model. It is based on the concept of gradient-based sampling, which involves sampling the data points based on their gradient values. GOSS works by down-sampling the large gradient data points while keeping all the small gradient data points.
      The main idea behind GOSS is that large gradient data points contribute more to the loss function and are therefore more important for the model training process. By down-sampling these large gradient data points, the training process can be accelerated without compromising the model's accuracy. On the other hand, small gradient data points are important for the generalization of the model, so they are kept in the training process.
      GOSS uses a two-step process for selecting the data points to be used in each iteration of the training process. In the first step, GOSS selects a small random subset of the data points, including all of the small gradient data points. In the second step, it further selects a larger subset of data points, including a portion of the large gradient data points. The ratio between the small and large subsets can be adjusted to control the trade-off between training speed and accuracy.

- **Exclusive Feature Bundling:** is a technique used in LightGBM to improve the accuracy of the model and reduce its computational complexity. It works by combining related features into bundles, which are then treated as a single feature during the training process. The main idea behind Exclusive Feature Bundling is to identify groups of features that are highly correlated with each other and bundle them together to reduce the number of features that the model has to learn from. By doing this, it can reduce the dimensionality of the feature space, making the model more efficient and easier to interpret. The technique of Exclusive Feature Bundling is similar to feature hashing, but with some key differences. Feature hashing creates a fixed number of hash buckets to map features to, which can lead to collisions and loss of information. Exclusive Feature Bundling, on the other hand, groups together only those features that are highly correlated with each other, and ensures that each feature is included in only one bundle. This eliminates the problem of collisions and preserves more of the original information. Exclusive Feature Bundling can be used with any type of data, including numerical, categorical, and text data. It can also be combined with other feature engineering techniques, such as one-hot encoding and feature scaling.

    - LightGBM splits the tree leaf-wise as opposed to other boosting algorithms that grow tree level-wise. It chooses the leaf with maximum delta loss to grow. Since the leaf is fixed, the leaf-wise algorithm has lower loss compared to the level-wise algorithm. Leaf-wise tree growth might increase the complexity of the model and may lead to overfitting in small datasets.
      Below is a diagrammatic representation of Leaf-Wise Tree Growth:



A histogram-based algorithm: is a method of processing data that involves dividing the data into discrete bins or buckets and creating a histogram to represent the distribution of the data. The histogram is a graphical representation of the frequency distribution of a set of continuous or discrete data.

In the context of machine learning, histogram-based algorithms are commonly used in decision trees and ensemble learning methods such as Gradient Boosted Decision Trees (GBDT). In these algorithms, the histogram is used to efficiently compute the splits in the decision tree.

Instead of calculating statistics for each data point, which can be computationally expensive, the data is first grouped into bins or buckets, and then the algorithm calculates the statistics for each bin. The number of bins and their sizes can significantly affect the performance of the algorithm.

- **Limitations:**
- **Loss of precision:** By grouping data into discrete bins, histogram-based algorithms can lose some precision in the data. The size and number of bins used can have a significant impact on the performance of the algorithm, and if the bins are not chosen carefully, the algorithm may miss important patterns in the data.
- **Limited flexibility:** Histogram-based algorithms are often less flexible than other methods, as the number and size of the bins need to be set prior to training. This can make it challenging to adapt the algorithm to new datasets or to capture complex patterns in the data.
- **Sensitivity to outliers:** Histogram-based algorithms can be sensitive to outliers, as they can skew the distribution of the data and affect the binning process. This can result in the algorithm misrepresenting the underlying patterns in the data.
- **Difficulty with high-dimensional data:** Histogram-based algorithms can struggle with high-dimensional data, as the number of bins required to effectively represent the data can grow exponentially with the number of dimensions. This can lead to issues with overfitting, where the algorithm fits too closely to the training data and does not generalize well to new data.

- **Model parameters:**

1. **max_depth :** It sets a limit on the depth of tree. The default value is 20. It is effective in controlling over fitting.
2. **categorical_feature :** It specifies the categorical feature used for training model.
3. **bagging_fraction :** It specifies the fraction of data to be considered for each iteration.
4. **num_iterations :** It specifies the number of iterations to be performed. The default value is 100.
5. **num_leaves :** It specifies the number of leaves in a tree. It should be smaller than the square of *max_depth*.
6. **max_bin :** It specifies the maximum number of bins to bucket the feature values.
7. **min_data_in_bin :** It specifies minimum amount of data in one bin.
8. **task :** It specifies the task we wish to perform which is either train or prediction. The default entry is *train*. Another possible value for this parameter is *prediction.*
9. **feature_fraction** : It specifies the fraction of features to be considered in each iteration. The default value is one.

- **Here is a summary of how LightGBM works:**

LightGBM is a gradient boosting framework that is designed to be efficient, scalable, and accurate for large-scale machine learning tasks. It works by building an ensemble of decision trees, where each tree is trained to predict the residuals (i.e., the differences between the actual and predicted values) of the previous trees in the ensemble.

The key features of LightGBM include:

1. Gradient-based One-Side Sampling (GOSS): This technique involves sampling data points based on their gradient values, which improves the efficiency of the training process by down-sampling large gradient data points while keeping small gradient data points.
2. Exclusive Feature Bundling: This technique involves grouping together highly correlated features into bundles, which reduces the dimensionality of the feature space and improves the efficiency of the model.
3. Histogram-based Gradient Boosting: LightGBM uses a histogram-based approach for computing gradients, which reduces the computational cost of computing the gradients and speeds up the training process.
4. Leaf-wise Tree Growth: LightGBM grows trees in a leaf-wise manner, where each leaf is grown by finding the best split among all the features, which leads to better accuracy and faster convergence.

Gradient Boosting vs AdaBoost vs XGBoost vs CatBoost vs LightGBM:

- **Gradient Boosting:**
  is the boosting algorithm that works on the principle of the stagewise addition method, where multiple weak learning algorithms are trained and a strong learner algorithm is used as a final model from the addition of multiple weak learning algorithms trained on the same dataset.

  In the gradient boosting algorithm, the first weak learner will not be trained on the dataset, it will simply return the mean of the particular column, and the residual for output of the first weak learner algorithm will be calculated which will be used as output or target column for next weak learning algorithm which is to be trained. Following the same pattern, the second weak learner will be trained and the residuals will be calculated which will be used as an output column again for the next weak learner, this is how this process will continue until we reach zero residuals.

  In gradient boosting the dataset should be in the form of numerical or categorical data and the loss function using which the residuals are calculated, should be differentiable at all points

- **XGBoost:**

  The main difference between Gradient Boosting and XGBoost is that XGbost uses a regularization technique in it. In simple words, it is a regularized form of the existing gradient-boosting algorithm.

  Due to this, XGBoost performs better than a normal gradient boosting algorithm and that is why it is much faster than that also. It also performs better when there is a presence of numerical and categorical features in the dataset.

- **AdaBoost:**

  is a boosting algorithm, which also works on the principle of the stagewise addition method where multiple weak learners are used for getting strong learners. Unlike Gradient Boosting in XGBoost, the alpha parameter I calculated is related to the errors of the weak learner, here the value of the alpha parameter will be indirectly proportional to the error of the weak learner.

  Once the alpha parameter is calculated, the weightage will be given to the particular weak learners, here the weak learner that are doing mistakes will get more weightage to fill out the gap in error and the weak learners that are already performing well will get fewer weights as they are already a good model.

- **CatBoost:**
  the main difference that makes it different and better than others is the growing of decision trees in it. In CatBoost the decision trees which is grown are symmetric.

  CatBoost is a boosting algorithm that performs exceptionally very well on categorical datasets other than any algorithm in the field of machine learning as there is a special type of method for handling categorical datasets. In CatBoost, the categorical features are encoded on the basis of the output columns. So while training or encoding the categorical features, the weightage of the output column will also be considered which makes it higher accurate on categorical datasets.

- **LightGBM:**
  is also a boosting algorithm, which means Light Gradient Boosting Machine. It is used in the field of machine learning. In LightGBM decision trees are grown leaf wise meaning that at a single time only one leaf from the whole tree will be grown.

  LightGBM also works well on categorical datasets and it also handles the categorical features using the binning or bucketing method. To work with categorical features in LightGBM we have converted all the categorical features in the category data type. Once done, there will be no need to handle categorical data as it will handle it automatically.

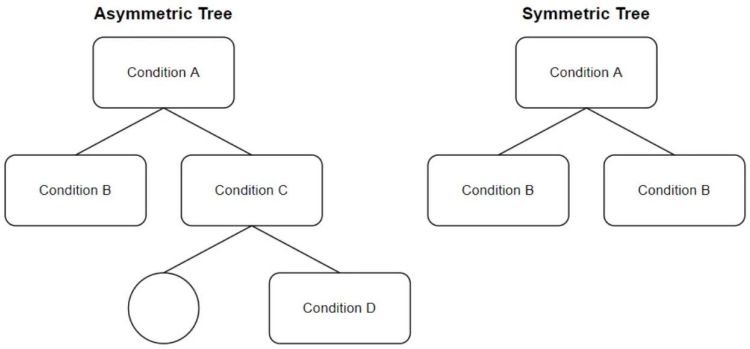
  In LightGBM, the sampling of the data while training the decision tree is done by the method known as GOSS. In this method, the variance of all the data samples is calculated and sorted in descending order. Data samples having low variance are already performing well, so there will be less weightage given to the samples having low variance while sampling the dataset.

- ## Which to Use?
  If you think that there is a need for regularization according to your dataset, then you can definitely use XGBoost, If you want to deal with categorical data, then CatBoost and LightGBM perform very well on those types of datasets. If you need more community support for the algorithm then use algorithms which was developed years back.

- **Catboost vs. LightGBM vs. XGBoost Characteristics**

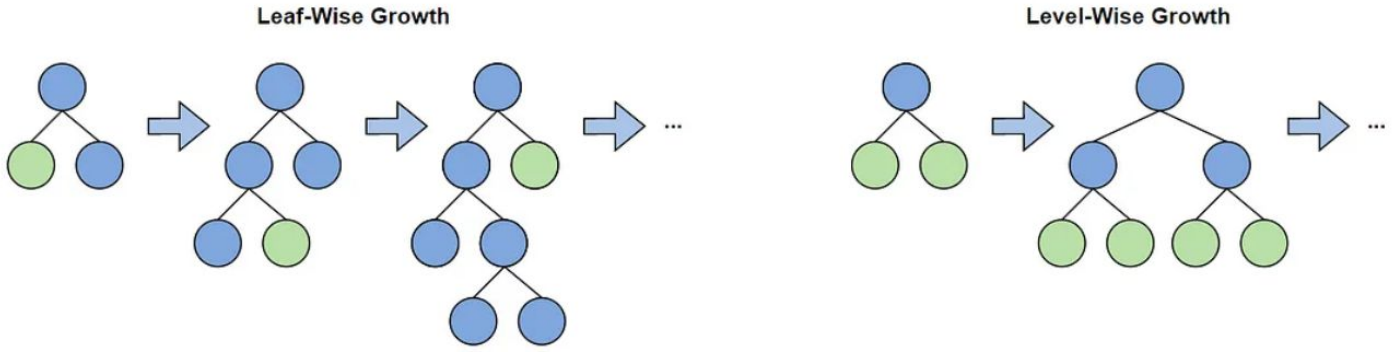
	CatBoost	LightGBM	XGBoost
Developer	Yandex	Microsoft	DMLC
Release Year	2017	2016	2014
Tree Symmetry	Symmetric	Asymmetric Leaf-wise tree growth	Asymmetric Level-wise tree growth
Splitting Method	Greedy method	Gradient-based One-Side Sampling (GOSS)	Pre-sorted and histogram-based algorithm
Type of Boosting	Ordered	-	-
Numerical Columns	Support	Support	Support
Categorical Columns	Support Perform one-hot encoding (default) Transforming categorical to numerical columns by border, bucket, binarized target mean value, counter methods available	Support, but must use numerical columns Can interpret ordinal category	Support, but must use numerical columns Cannot interpret ordinal category, users must convert to one-hot encoding, label encoding or mean encoding
Text Columns	Support Support Bag-of-Words, Naïve-Bayes or BM-25 to calculate numerical features from text data	Do not support	Do not support
Missing values	Handle missing value Interpret as NaN (default) Possible to interpret as error, or processed as minimum or maximum values	Handle missing value Interpret as NaN (default) or zero Assign missing values to side that reduces loss the most in each split	Handle missing value Interpret as NaN (tree booster) or zero (linear booster) Assign missing values to side that reduces loss the most in each split

- In CatBoost, symmetric trees, or balanced trees, refer to the splitting condition being consistent across all nodes at the same depth of the tree. LightGBM and XGBoost, on the other hand, results in asymmetric trees, meaning splitting condition for each node across the same depth can differ.



- For symmetric trees, this means that the splitting condition must result in the lowest loss across all nodes of the same depth. Benefits of balanced tree architecture include faster computation and evaluation and control overfitting.

- Even though LightGBM and XGBoost are both asymmetric trees, LightGBM grows leaf-wise while XGBoost grows level-wise. To put it simply, we can think of LightGBM as growing the tree selectively, resulting in smaller and faster models compared to XGBoost.



- **Splitting Method:**

  - In CatBoost, a greedy method is used such that a list of possible candidates of feature-split pairs are assigned to the leaf as the split and the split that results in the smallest penalty is selected.
  - In LightGBM, Gradient-based One-Side Sampling (GOSS) keeps all data instances with large gradients and performs random sampling for data instances with small gradients. Gradient refers to the slope of the tangent of the loss function. Data points with larger gradients have higher errors and would be important for finding the optimal split point, while data points with smaller gradients have smaller errors and would be important for keeping accuracy for learned decision trees. This sampling technique results in lesser data instances to train the model and hence faster training time.
  - In XGBoost, the pre-sorted algorithm considers all feature and sorts them by feature value. After which, a linear scan is done to decide the best split for the feature and feature value that results in the most information gain. The histogram-based algorithm works the same way but instead of considering all feature values, it groups feature values into discrete bins and finds the split point based on the discrete bins instead, which is more efficient than the pre-sorted algorithm although still slower than GOSS.

- **Performance Comparison:** XGBoost and LightGBM yield similar performance, with CatBoost and LightGBM performing much faster than XGBoost, especially for larger datasets.

- Improving Accuracy, Speed, and Controlling Overfitting

	CatBoost	LightGBM	XGBoost
Parameters to tune	*iterations*: number of trees *depth*: depth of tree *min_data_in_leaf*: control depth of tree	*num_leaves*: value should be less than 2^max_depth *min_data_in_leaf*: control depth of tree *max_depth*: depth of tree	*n_estimators*: number of trees *max_depth*: depth of tree *min_child_weight*: control depth of tree
Parameters for better accuracy		*max_bin*: maximum number of bins feature values will be bucketed in *num_leaves* Use bigger training data	
Parameters for faster speed	*subsample*: fraction of number of instances used in a tree *rsm*: random subspace method; fraction of number of features used in a split selection *iterations* *sampling_frequency*: frequency to sample weights and objects when building trees	*feature_fraction*: fraction of number of features used in a tree *bagging_fraction*: fraction of number of instances used in a tree *bagging_freq*: frequency for bagging *max_bin* *save_binary*: indicator to save dataset to binary file Use parallel learning	*colsample_bytree*: fraction of number of features used in a tree *subsample*: fraction of number of instances used in a tree *n_estimators*
Parameters to control overfitting	*early_stopping_rounds*: stop training after specified number of iterations since iteration with optimal metric value *od_type*: type of overfitting detector *learning_rate*: learning rate for reducing gradient step *depth* *l2_leaf_reg*: regularization parameter	*max_bin* *num_leaves* *max_depth* *bagging_fraction* *bagging_freq* *feature_fraction* *lambda_l1 / lambda_l2 / min_gain_to_split* Use bigger training data Regularization	*learning_rate* *gamma*: regularization parameter, higher gamma for more regularization *max_depth* *min_child_weight* *subsample*