## **Machine Learning Algorithms:**

Machine Learning algorithm is an evolution of the regular algorithm. It makes your programs smarter, by allowing them to automatically learn from the data you provide. The algorithm is mainly divided into:

* Training Phase
* Testing phase

**Training Phase**

You take a randomly selected specimen of apples from the market **training data**, make a table of all the physical characteristics of each apple, like color, size, shape, grown in which part of the country, sold by which vendor, etc. features, along with the sweetness, juiciness, ripeness of that apple **output variables**.

**Testing Phase**

Next time when you go shopping, you will measure the characteristics of the apples which you are purchasing **test data** and feed it to the Machine Learning algorithm the algorithm may internally use the rules, similar to the one you manually wrote earlier (for e.g., a **decision tree**).

**Linear Regression Algorithm.**

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous and has a constant slope. It’s used to predict values within a continuous range, (e.g. sales, price) rather than trying to classify them into categories (e.g. cat, dog). Linear regression uses traditional slope-intercept form, where m and b are the variables our algorithm will try to “learn” to produce the most accurate predictions. X represents our input data and y represents our prediction.

**y=mx+b**

**X:** Dependent Variable

**Y:** Independent Variable

**Cost Function**

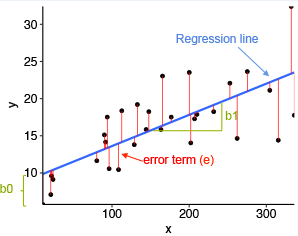
Let’s use [MSE (L2)](https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#mse) as our cost function. MSE measures the average squared difference between an observation’s actual and predicted values. The output is a single number representing the cost, or score, associated with our current set of weights. Our goal is to minimize MSE to improve the accuracy of our model.

**Math**

Given our simple linear equation y=mx+b*y=mx+b*, we can calculate MSE as:

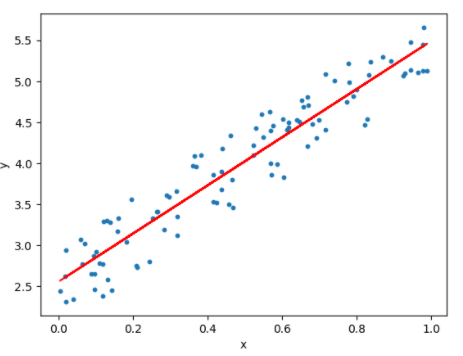
MSE=1N∑i=1n(yi−(mxi+b))2

* N is the total number of observations (data points)
  + 1N∑ni=1*1N∑i=1n* is the mean
* Yi is the actual value of an observation and mxi+b*mxi+b* is our prediction



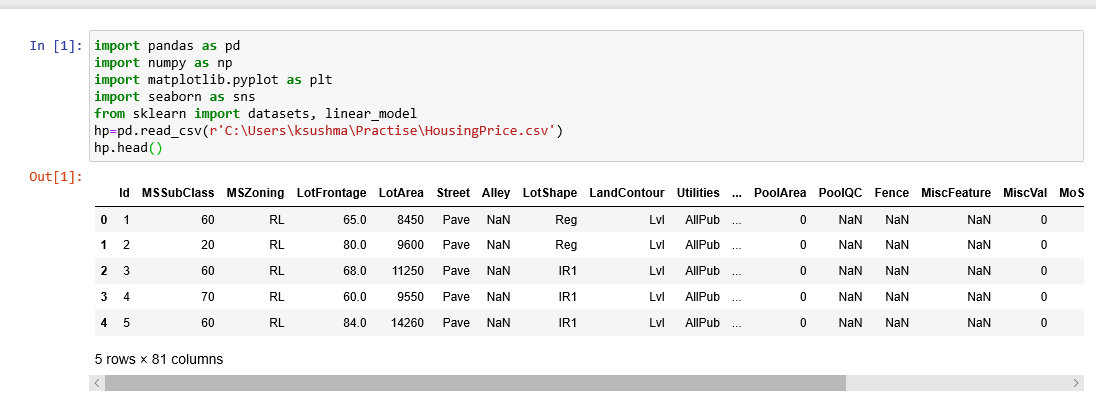
**Gradient descent**

To minimize MSE we use [Gradient Descent](https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html) to calculate the gradient of our cost function. Gradient descent consists of looking at the error that our weight currently gives us, using the derivative of the cost function to find the gradient (The slope of the cost function using our current weight), and then changing our weight to move in the direction opposite of the gradient. We need to move in the opposite direction of the gradient since the gradient points up the slope instead of down it, so we move in the opposite direction to try to decrease our error.



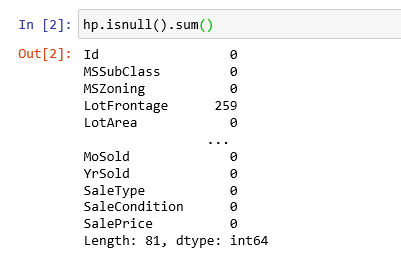
**Example:** Let's take an example of Housing Price Dataset from Kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy



**EDA Analysis:**

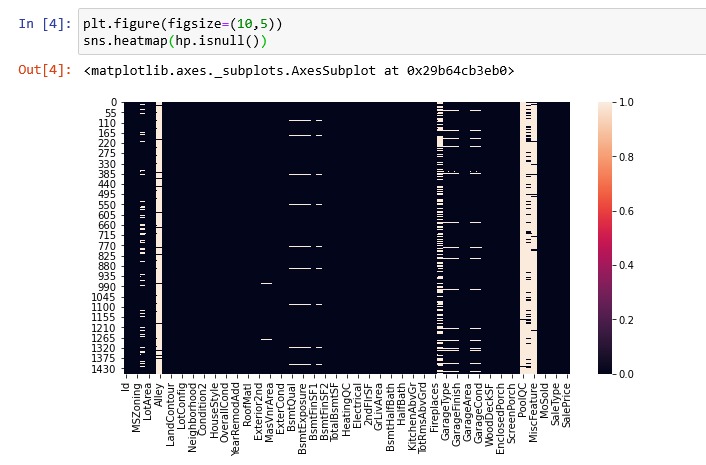
**Step 2:** Check the Null Values if any in the dataset.



To Know Number of Columns and rows in a Dataset;

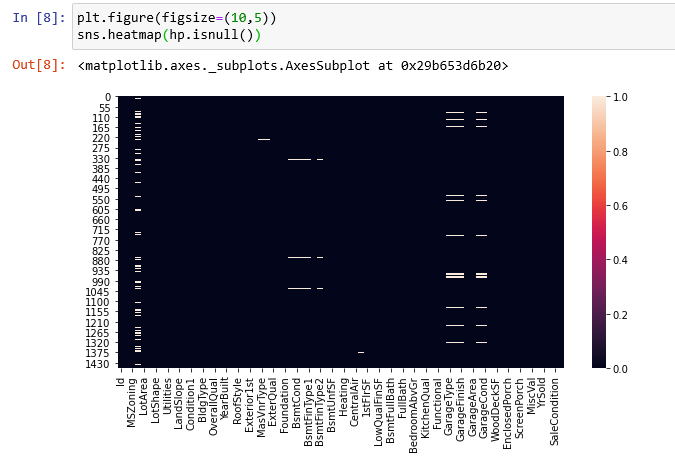


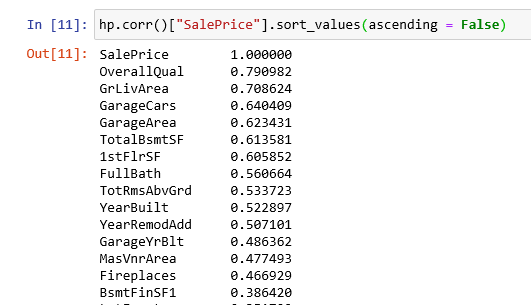
**Step 3:** To represent **Null** Values in a pictorial format (Heatmap);

In this visualization, few columns (Alley, PoolQC, MisFeature, Fence, FireplaceQu) have null values around 90%. For this predicition we can drop these columns.



**Step 4:** After dropping these columns we can see null values still exists in our dataset. So, these null values and missing values should be handled.

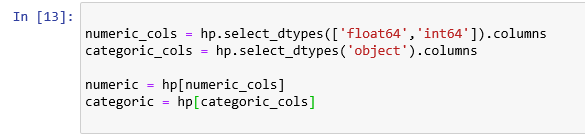
**Step 5:** Finding Correlation for target variable and rest of the columns.



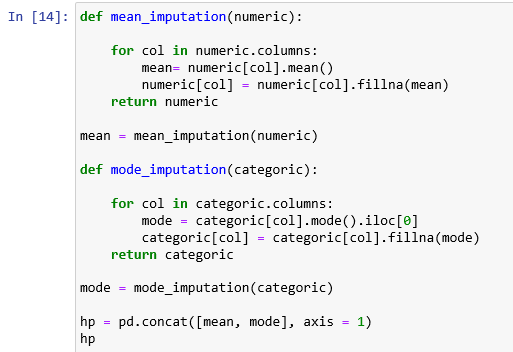
Here **OverallQual** has the highest correlation amongst all columns.

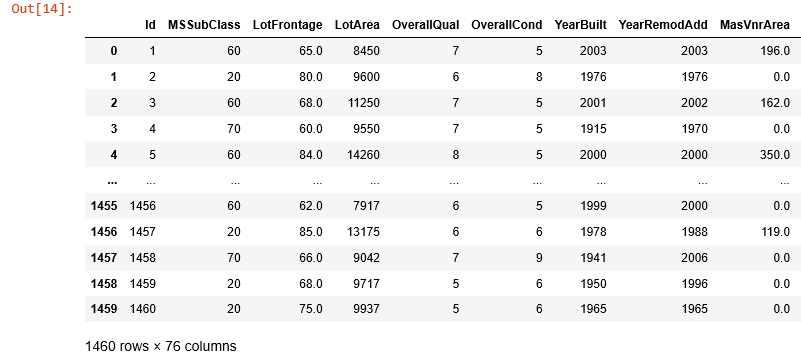
**Step 6:** Handling null values by replacing them with **Mean / Median/ Mode:**

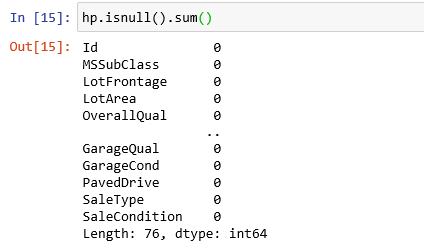
To handle Null values, need to separate Numerical and data Categorical values from the dataset.



Now Replacing Null values with Mean and Mode:

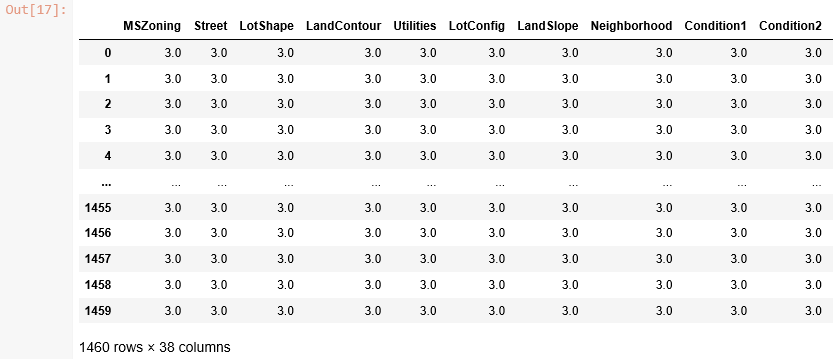
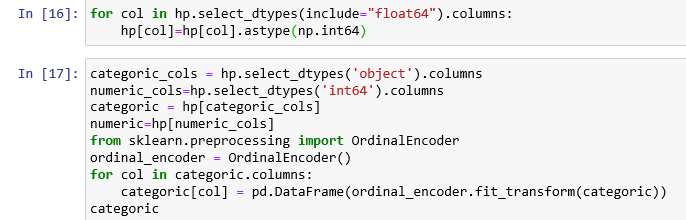


Now again, finding if any missing values exists:



Missing values were handled and we can see the count is zero.

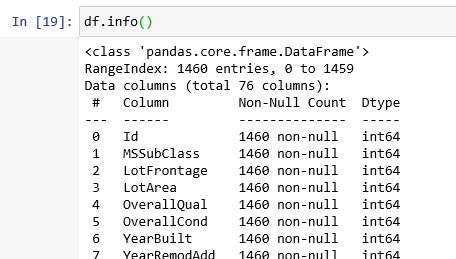
**Step 7:** Convert Categorical columns with 0’s and 1’s, if any string values exists in dataset.



Now joining both numerical and categorical, so that in updated Dataset we will have only data with datatypes of int and float.

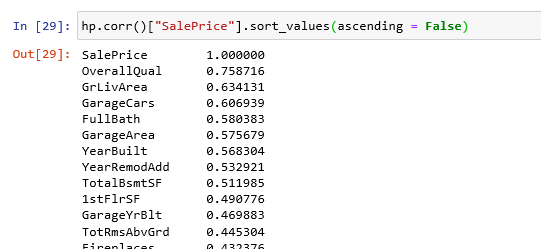


To see the datatypes for validation if datatypes of int and float only exists,



**Correlation**

**Step 8:** Checking the Correlation after EDA Analysis:



Now, Correlation has been changed and we need to take the nearest value which exists for target and need to Predict the Regression on target variable and OverallQual.

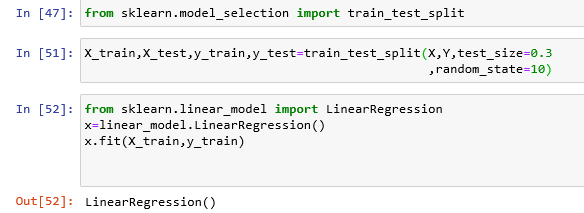
So, taking X as QverallQual from concatenated dataset and target from Historical Dataset.



**Train and Test**

**Step 9:** Now, we need to train and test the data. So, import package from library.

Training the model with X\_train and y\_train and applying Linear Regression.



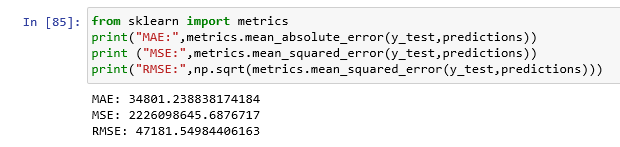
Note: If any reshape issue use code below code and run.

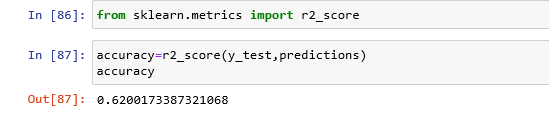
**X=X.values.reshape(len(X),1)**  
**Y=Y.values.reshape(len(Y),1)**

**Step 10:** Now checking for Best fit line from the datapoints.



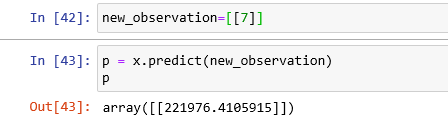
**Step 11:** Checking the errors and that has reduced to get the Regression line.

**Step 12:** Finding the accuracy of our prediction by using r2 score



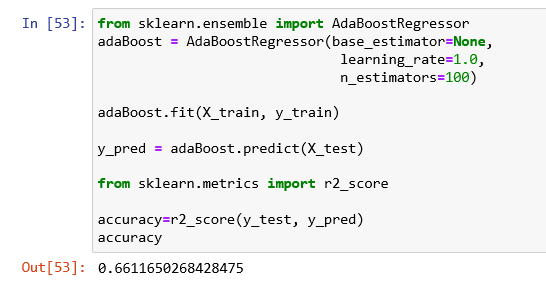
We got 62% Accuracy for our Prediction.

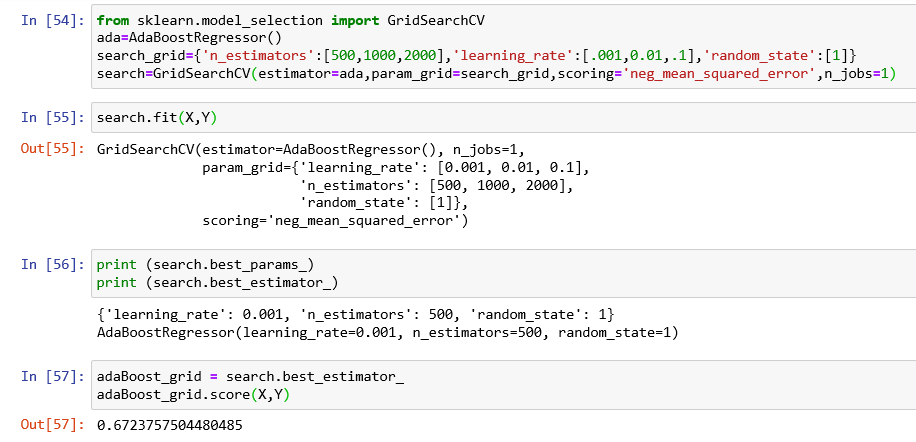
**Step 13:** Check the output for X using new observation to know the accuracy and the prediction is correct or not.



So, the value with array is nearest to the Target variable (SalePrice).

Using Adaboost and GridSearchCV to improve accuracy.





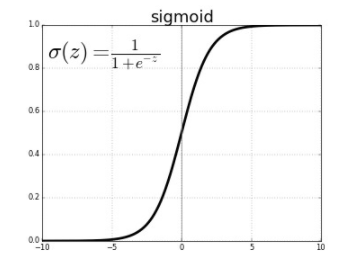
**Logistic Regression**

Logistic regression is a classification algorithm, used when the value of the target variable is categoricalin nature. Logistic regression is most commonly used when the data in question has binary output, so when it belongs to one class or another, or is either a 0 or 1.

Remember that classification tasks have discrete categories, unlike regressions tasks.

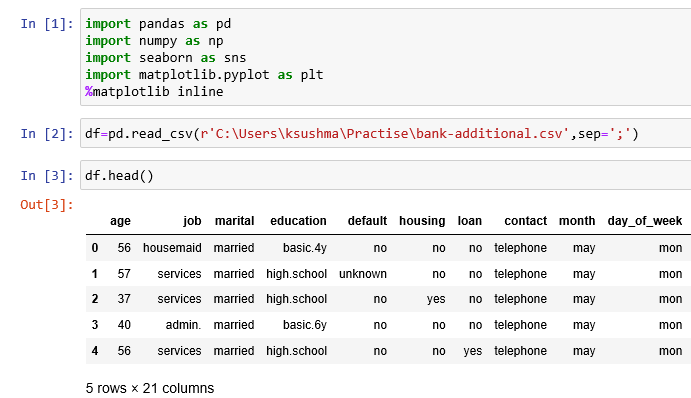
## **The Sigmoid Function**

The sigmoid function/logistic function is a function that resembles an “S” shaped curve when plotted on a graph. It takes values between 0 and 1 and “squishes” them towards the margins at the top and bottom, labeling them as 0 or 1.

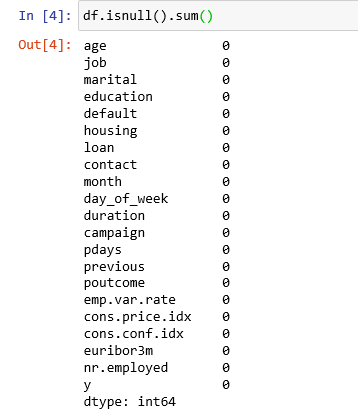


**Example:** Let's take an example of Bank Marketing Campaign Dataset from kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy

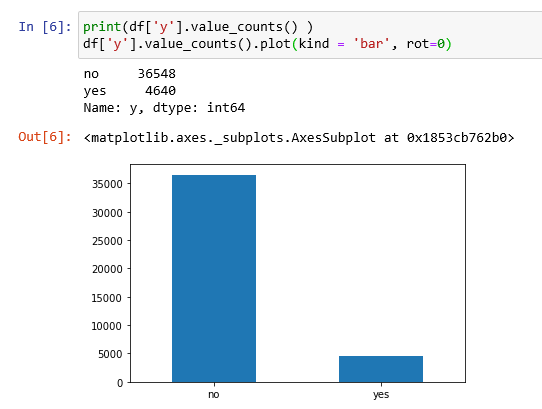
**Exploratory Data Analysis**

**Step 2:** Finding Missing Values



There are no missing values here. So, nothing to handle missing and null values.

**Step 3:** Check weather data is balanced or imbalanced.

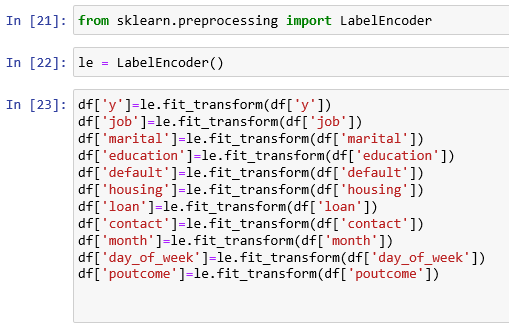


Data is imbalanced here; So, we need to balance the data by using RandomOverSampler.

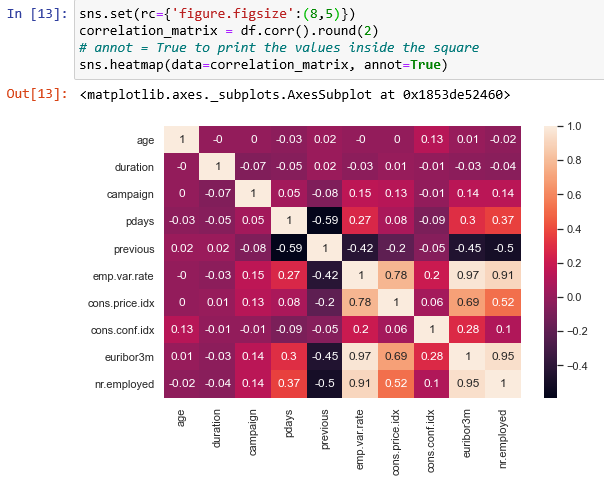


Now data is balanced and let's check the correlation.

**Step 4:** Convert Categorical data to 0’s and 1’s if any in the data set. In our dataset Categorical data exists. So, we need to convert them using Encoder.

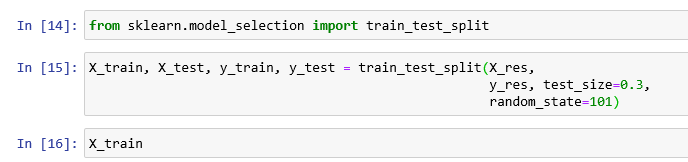


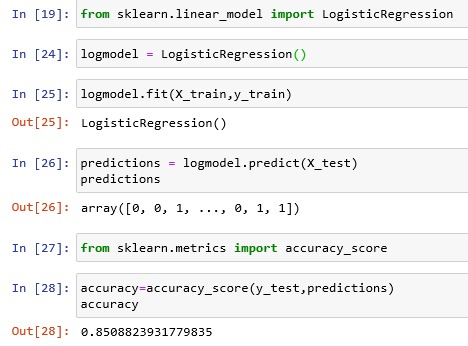
**Step 5:** Now check the correlation.



**Train and Test**

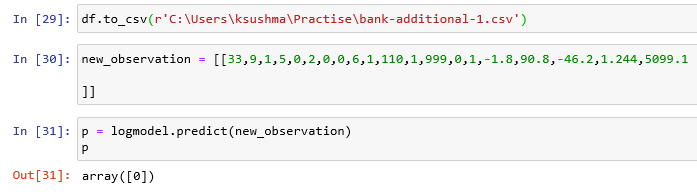
**Step 6:** Now Split the data to Train and test the data.

**Step 7:** Apply Logistic Regression and test the accuracy.



Output Predicted was 85% accuracy.

**Step 8:** Test the output with new observations and predict the value.

Output was correct with array [0]. Checking it from the target variable.

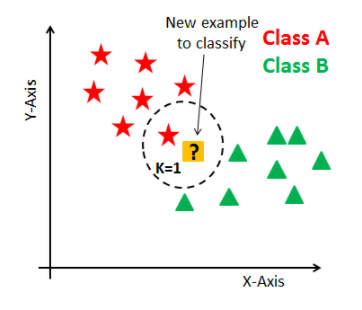
**KNN Algorithm**

K nearest neighbors or KNN Algorithm is a simple algorithm which uses the entire dataset in its training phase. Whenever a prediction is required for an unseen data instance, it searches through the entire training dataset for k-most similar instances and the data with the most similar instance is finally returned as the prediction.

K in KNN algorithm represents the number of nearest neighbor points which are voting for the new test data’s class.

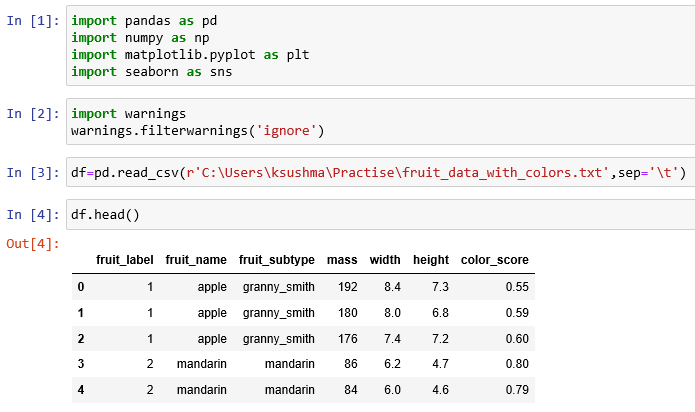
* If k=1, then test examples are given the same label as the closest example in the training set.
* If k=3, the labels of the three closest classes are checked and the most common (i.e., occurring at least twice) label is assigned, and so on for larger KS.

KNN is often used in simple recommendation systems, image recognition technology, and decision-making models. It is the algorithm companies like Netflix or Amazon use in order to recommend different movies to watch or books to buy.

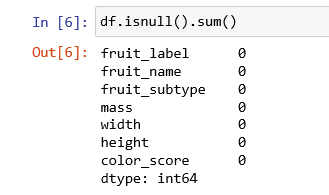


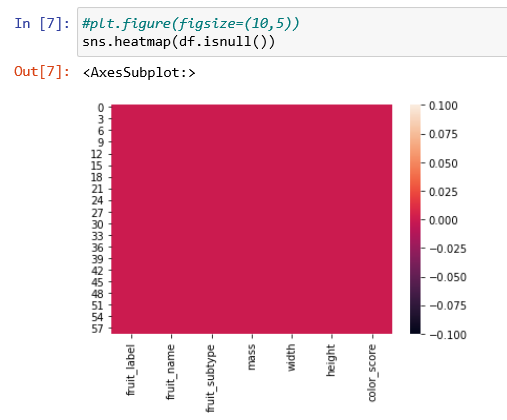
**Example:** Let's take an example of Fruits with colors Dataset from kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy

**EDA Analysis**

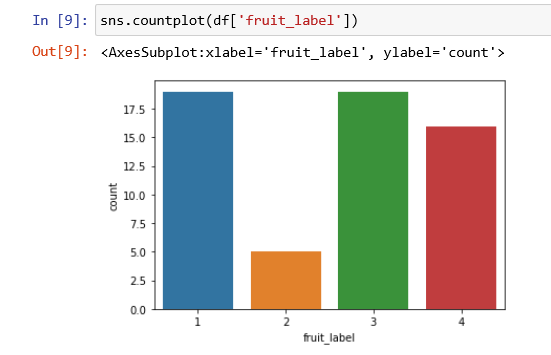
**Step 2:** Checking Null Values if any.





There is no null Values here and nothing to handle.

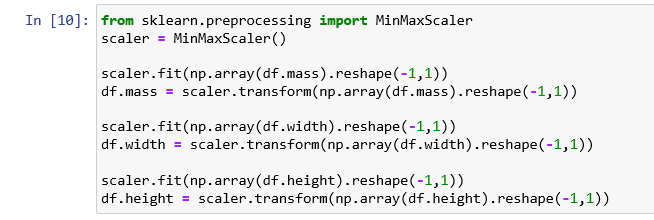
**Step 3:** Check weather data is balanced or imbalanced.

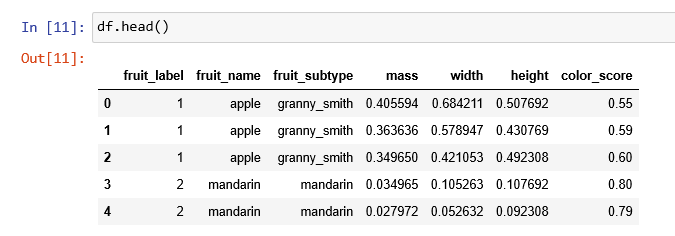


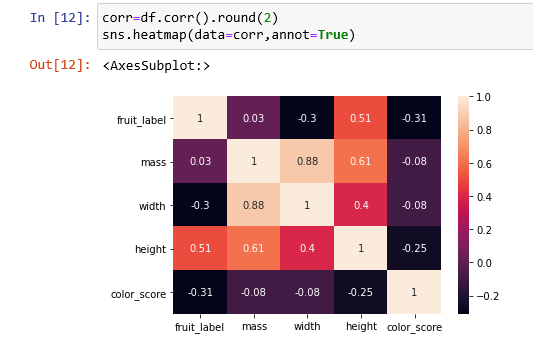
Data is almost balanced here which is from the historical data.

**Step 4:** Scaling the data using reshape to fit the data and should be in one format.

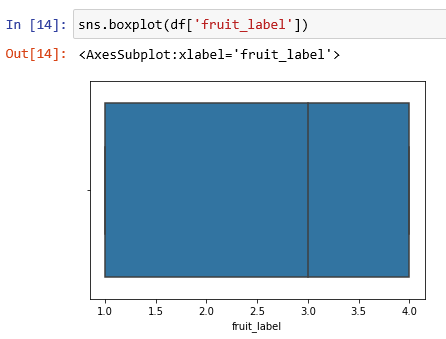
Here we used MinMaxScaler function to scale the data.

 After scaling the data metrics with mass, weight and height; Data changes to float values with all in same format.

**Step 5:** Check the Correlation.



**Step 6:** Checking for Outliers ----- No Outliers here.

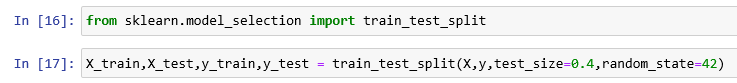


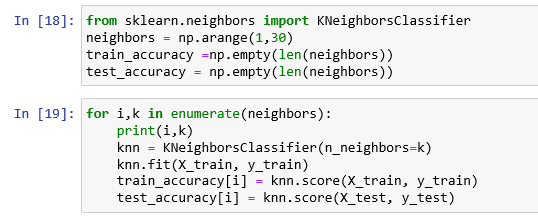
**Step 7:** Defining X and Y variables



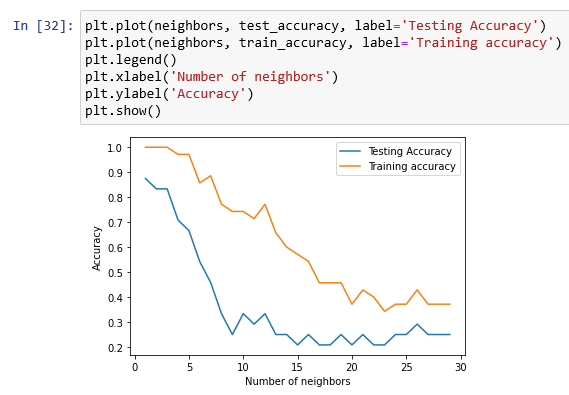
X with all Numerical columns and Y with Target Variable.

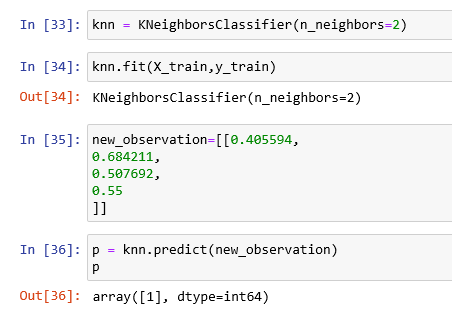
**Step 8:** Train and Test

Creating KNN Classifier and enumerating the distance from point to neighbors.



Pictorial Format: Representing train and test accuracies in one graph





Now, consider neighbor k value as 2 because the decrease and increasing of values is between 0.2 to 0.3 range. Validating the output by giving X values in the list and the output was correct with array[1].

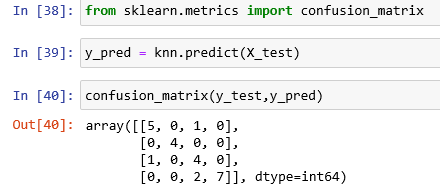
To find best K value, we can use GridSearchCV.



Checking the accuracy and it’s 83%.

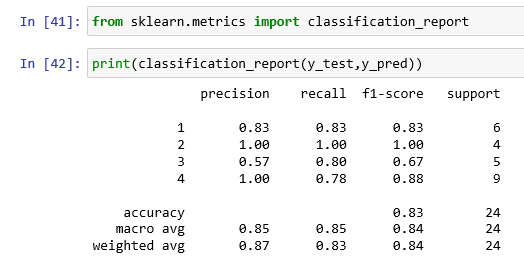


**Step 9:** Confusion Matrix



Checking the confusion matrix with prediction and y\_test values.

Step 10: Checking the classification for prediction with precision, recall and f1-score.



**Support Vector Machine SVM**

Support Vector Machine (SVM) is a supervised [machine learning algorithm](https://courses.analyticsvidhya.com/courses/introduction-to-data-science-2?utm_source=blog&utm_medium=understandingsupportvectormachinearticle) which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes very well.

Support Vectors are simply the co-ordinates of individual observation. The SVM classifier is a frontier which best segregates the two classes (hyper-plane/ line).

You can look at [support vector machines](https://www.analyticsvidhya.com/blog/2014/10/support-vector-machine-simplified/?utm_source=blog&utm_medium=understandingsupportvectormachinearticle) and a few examples of its working here.

## **How does it work?**

Above, we got accustomed to the process of segregating the two classes with a hyper-plane. Now the burning question is “How can we identify the right hyper-plane?”. Don’t worry, it’s not as hard as you think!

Let’s understand:

* **Identify the right hyper-plane (Scenario-1):** Here, we have three hyper-planes (A, B and C). Now, identify the right hyper-plane to classify star and circle.  
  

You need to remember a thumb rule to identify the right hyper-plane: “Select the hyper-plane which segregates the two classes better”. In this scenario, hyper-plane “B” has excellently performed this job.

* **Identify the right hyper-plane (Scenario-2):** Here, we have three hyper-planes (A, B and C) and all are segregating the classes well. Now, how can we identify the right hyper-plane?



Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as **Margin**. Let’s look at the below Diagram:



Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is high chance of miss-classification.

* **Identify the right hyper-plane (Scenario-3): Hint**:Use the rules as discussed in previous section to identify the right hyper-plane



Some of you may have selected the hyper-plane **B** as it has higher margin compared to **A.** But here is the catch, SVM selects the hyper-plane which classifies the classes accurately prior to maximizing margin. Here, hyper-plane B has a classification error and A has classified all correctly. Therefore, the right hyper-plane is **A.**

* **Can we classify two classes (Scenario-4)?** Below, I am unable to segregate the two classes using a straight line, as one of the stars lies in the territory of other(circle) class as an outlier.   
  

As I have already mentioned, one star at other end is like an outlier for star class. The SVM algorithm has a feature to ignore outliers and find the hyper-plane that has the maximum margin. Hence, we can say, SVM classification is robust to outliers.  

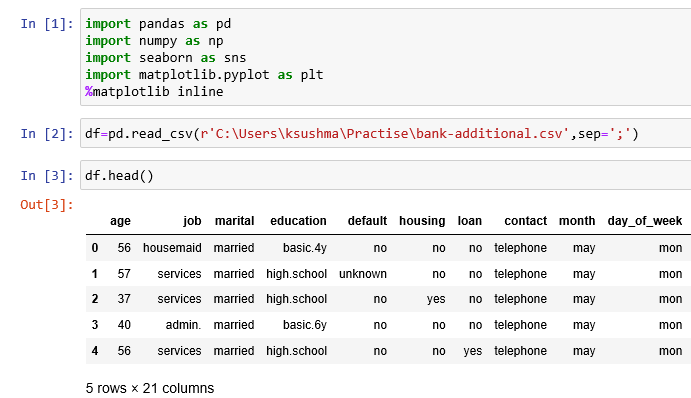

* **Find the hyper-plane to segregate to classes (Scenario-5):** In the scenario below, we can’t have linear hyper-plane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyper-plane.  
  

SVM can solve this problem. Easily! It solves this problem by introducing additional feature. Here, we will add a new feature z=x^2+y^2. Now, let’s plot the data points on axis x and z:  
  
In above plot, points to consider are:

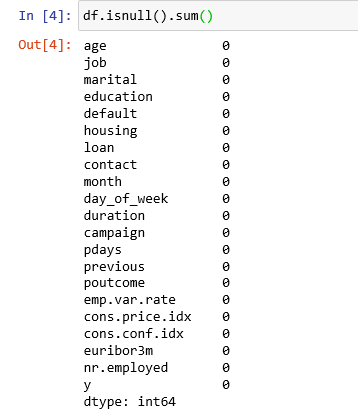
* + All values for z would be positive always because z is the squared sum of both x and y
  + In the original plot, red circles appear close to the origin of x and y axes, leading to lower value of z and star relatively away from the origin result to higher value of z.
* In the SVM classifier, it is easy to have a linear hyper-plane between these two classes. But another burning question which arises is, should we need to add this feature manually to have a hyper-plane. No, the SVM algorithm has a technique called the [kernel](https://en.wikipedia.org/wiki/Kernel_method) trick. The SVM kernel is a function that takes low dimensional input space and transforms it to a higher dimensional space i.e. it converts not separable problem to separable problem. It is mostly useful in non-linear separation problem. Simply put, it does some extremely complex data transformations, then finds out the process to separate the data based on the labels or outputs you’ve defined.

**Example:** Let's take an example of Bank Marketing Campaign Dataset from kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy

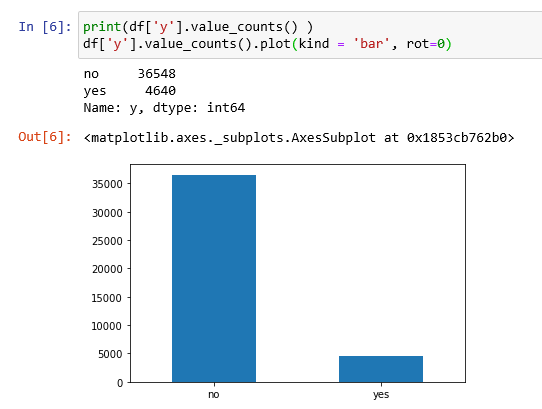
**Exploratory Data Analysis**

**Step 2:** Finding Missing Values



There are no missing values here. So, nothing to handle missing and null values.

**Step 3:** Check weather data is balanced or imbalanced.

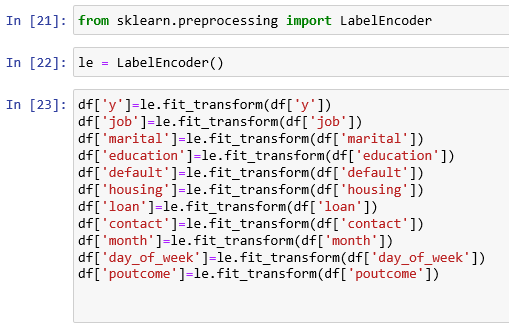


Data is imbalanced here; So, we need to balance the data by using RandomOverSampler.

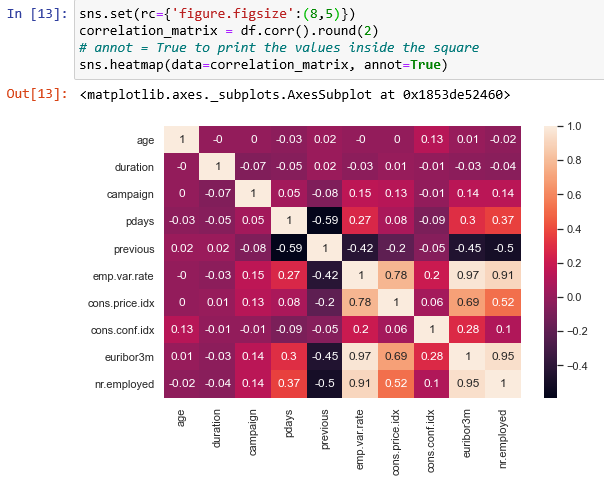


Now data is balanced and let's check the correlation.

**Step 4:** Convert Categorical data to 0’s and 1’s if any in the data set. In our dataset Categorical data exists. So, we need to convert them using Encoder.

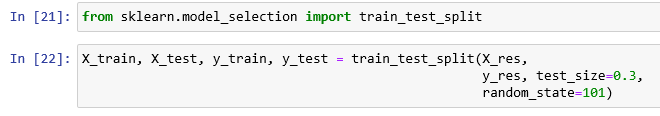


**Step 5:** Now check the correlation.

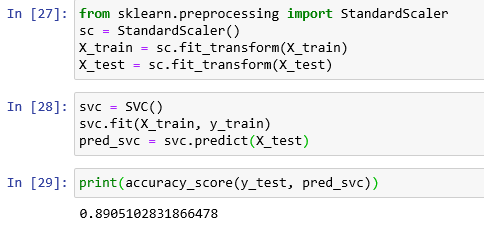


**Train and Test**

**Step 6:** Now Split the data to Train and test the data.

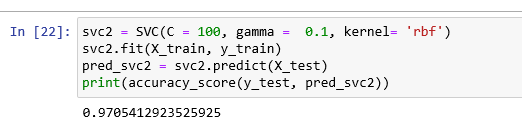


Import Standard Scalar from sklearn and scale X\_test and X\_train data to a range in one format.

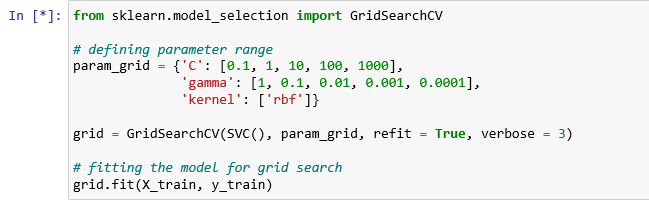


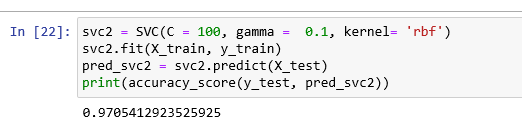
Predicted output with 89% accuracy.

After applying **c** and **gamma** values, accuracy is 97%



To find best **c** and **gamma** values, we can use GridSearchCV

Found best params and finding accuracy.



Increased accuracy from 89 to 97% after sing GridSearchCV.

**Decision Tree**

Decision Trees are a type of Supervised Machine Learning (that is you explain what the input is and what the corresponding output is in the training data) where the data is continuously split according to a certain parameter. The tree can be explained by two entities, namely decision nodes and leaves. The leaves are the decisions or the final outcomes. And the decision nodes are where the data is split.

## **Basic Decision Tree Terminologies**

* **Parent and Child Node:** A node that gets divided into sub-nodes is known as Parent Node, and these sub-nodes are known as Child Nodes. Since a node can be divided into multiple sub-nodes, therefore a node can act as a parent node of numerous child nodes
* **Root Node:** The top-most node of a decision tree. It does not have any parent node. It represents the entire population or sample
* **Leaf / Terminal Nodes:** Nodes that do not have any child node are known as Terminal/Leaf Nodes

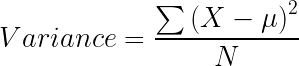
**Node splitting, or simply splitting, is the process of dividing a node into multiple sub-nodes to create relatively pure nodes.** There are multiple ways of doing this, which can be broadly divided into two categories based on the type of target variable:

* Continuous Target Variable
  + Reduction in Variance
* Categorical Target Variable
  + Gini Impurity
  + Information Gain
  + Chi-Square

## **Decision Tree Splitting Methods:**

## **Reduction in Variance**

Reduction in Variance is a method for splitting the node used when the target variable is continuous, i.e., regression problems. It is so-called because it uses variance as a measure for deciding the feature on which node is split into child nodes.



Variance is used for calculating the homogeneity of a node. If a node is entirely homogeneous, then the variance is zero.

Here are the steps to split a decision tree using reduction in variance:

* For each split, individually calculate the variance of each child node
* Calculate the variance of each split as the weighted average variance of child nodes
* Select the split with the lowest variance
* Perform steps 1-3 until completely homogeneous nodes are achieved

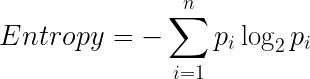
1. **Information Gain**

Now, what if we have a categorical target variable? Reduction in variation won’t quite cut it.

Well, the answer to that is Information Gain. Information Gain is used for splitting the nodes when the target variable is categorical. It works on the concept of the entropy and is given by:

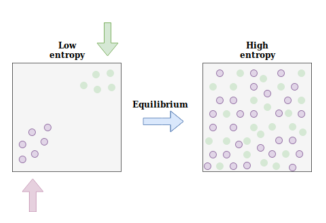
*Information Gain= 1 - Entropy*

Entropy is used for calculating the purity of a node. **Lower the value of entropy, higher is the purity of the node.** The entropy of a homogeneous node is zero. Since we subtract entropy from 1, the Information Gain is higher for the purer nodes with a maximum value of 1. Now, let’s take a look at the formula for calculating the entropy:



Steps to split a decision tree using Information Gain:

* For each split, individually calculate the entropy of each child node
* Calculate the entropy of each split as the weighted average entropy of child nodes
* Select the split with the lowest entropy or highest information gain
* Until you achieve homogeneous nodes, repeat steps 1-3

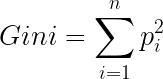


1. **Gini Impurity**

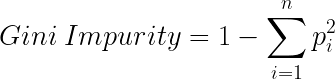
## Gini Impurity is a method for splitting the nodes when the target variable is categorical. It is the most popular and the easiest way to split a decision tree. The Gini Impurity value is:

*Gini Impurity= 1- Gini*

Gini is the probability of correctly labeling a randomly chosen element if it was randomly labeled according to the distribution of labels in the node. The formula for Gini is:



And Gini Impurity is:



Lower the Gini Impurity, higher is the homogeneity of the node. **The Gini Impurity of a pure node is zero.** Now, you might be thinking we already know about Information Gain then, why do we need Gini Impurity?

Gini Impurity is preferred to Information Gain because it does not contain logarithms which are computationally intensive.

Here are the steps to split a decision tree using Gini Impurity:

* Similar to what we did in information gain. For each split, individually calculate the Gini Impurity of each child node
* Calculate the Gini Impurity of each split as the weighted average Gini Impurity of child nodes
* Select the split with the lowest value of Gini Impurity
* Until you achieve homogeneous nodes, repeat steps 1-3

1. **Chi-Square**

Chi-square is another method of splitting nodes in a decision tree for datasets having categorical target values. It can make two or more than two splits. It works on the statistical significance of differences between the parent node and child nodes.

Chi-Square value is:

chi square decision tree

Here, the *Expected* is the expected value for a class in a child node based on the distribution of classes in the parent node, and *Actual* is the actual value for a class in a child node.

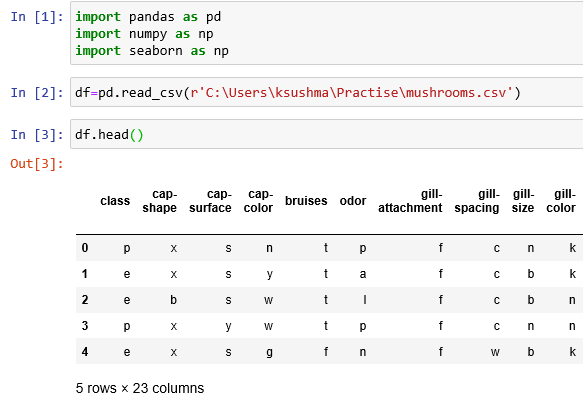
The above formula gives us the value of Chi-Square for a class. Take the sum of Chi-Square values for all the classes in a node to calculate the Chi-Square for that node. Higher the value, higher will be the differences between parent and child nodes, i.e., higher will be the homogeneity.

Here are the steps to split a decision tree using Chi-Square:

* For each split, individually calculate the Chi-Square value of each child node by taking the sum of Chi-Square values for each class in a node
* Calculate the Chi-Square value of each split as the sum of Chi-Square values for all the child nodes
* Select the split with higher Chi-Square value
* Until you achieve homogeneous nodes, repeat steps 1-3

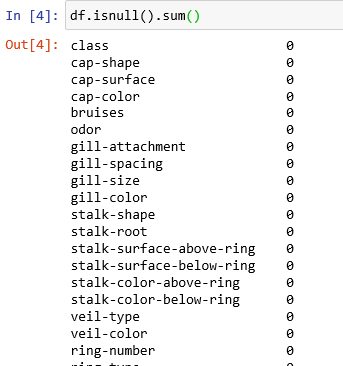
**Example:** Let's take an example of Mushroom Dataset from Kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy

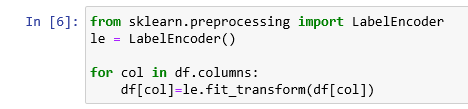


**EDA Analysis**

**Step 2:** Checking for null Values if any;

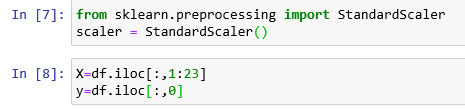


**Step 3:** No Null Values found here. But the data is in categorical format, so converting categorical data Numerical using Label Encoder.



Here the whole data is in categorical format we need to convert them. For that, we are using for loop to convert whole data.

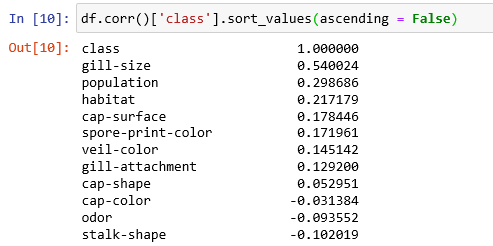
**Step 4:** Scaling the data and splitting X and Y values



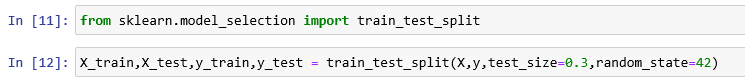
Here by using iloc function, and providing values in it for finding positions and diving X and Y values.

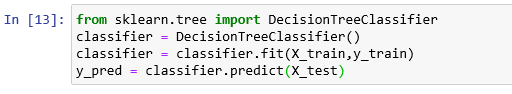
By using head (), we can see all the data, and here the entire data converted to numerical

**Step 5:** Finding the correlation.

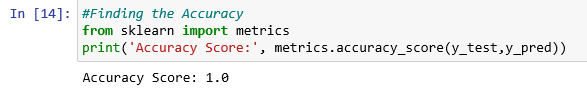


**Step 6:** Train and Test

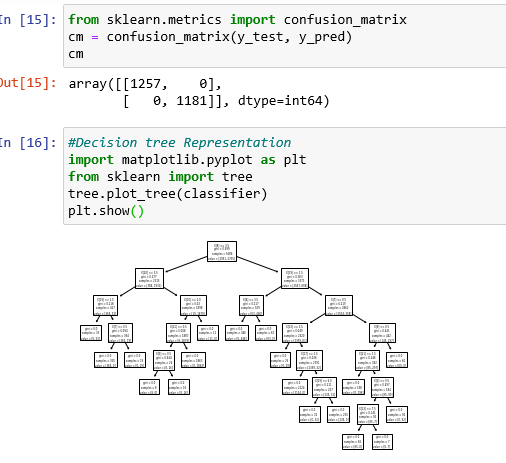
**Step 7:** Importing Decision Tree Classifier and Predicting y\_test



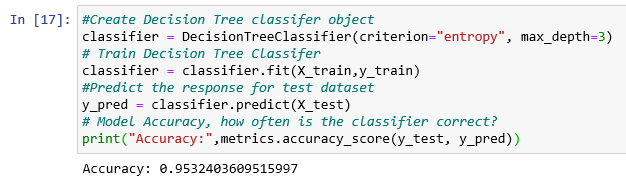
**Step 8:** Finding the Accuracy

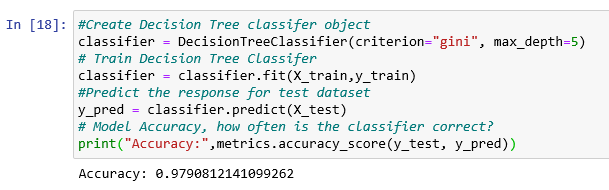


**Step 9:** Checking Confusion Matrix and see the decision tree using plots.



**Step 10:** Using entropy, to find the accuracy and providing parameters.

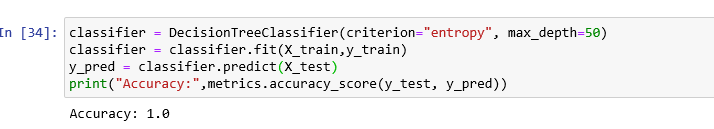
**Step 11:** Using entropy, to find the accuracy and providing parameters.



To find which parameters increases the accuracy of the model can be done by using GridsearchCV.

Predicted algorithm with best accuracy by applying **gini** and **entropy** parameters randomly. To find which is best out of them, we can use hyper tuning parameters.

Found best estimators and fit the train data.

Implemented using best estimators in Decision tree algorithm with best accuracy.

**Random Forest**

Random forest is a [supervised learning algorithm](https://builtin.com/data-science/supervised-learning-python). The "forest" it builds, is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result. One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems. Let's look at random forest in classification, since classification is sometimes considered the building block of machine learning.

Random forest has nearly the same hyperparameters as a decision tree or a bagging classifier. Fortunately, there's no need to combine a decision tree with a bagging classifier because you can easily use the classifier-class of random forest. With random forest, you can also deal with regression tasks by using the algorithm's regressor.

The hyperparameters in random forest are either used to increase the predictive power of the model or to make the model faster. Let's look at the hyperparameters of sklearns built-in random forest function.

## **IMPORTANT HYPERPARAMETERS**

**1. Increasing the predictive power**

Firstly, there is the **n\_estimators** hyperparameter, which is just the number of trees the algorithm builds before taking the maximum voting or taking the averages of predictions. In general, a higher number of trees increases the performance and makes the predictions more stable, but it also slows down the computation.

Another important hyperparameter is **max\_features,** which is the maximum number of features random forest considers to split a node. Sklearn provides several options, all described in the [documentation](http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html).

The last important hyperparameter is **min\_sample\_leaf.** This determines the minimum number of leafs required to split an internal node.

**2. Increasing the model's speed**

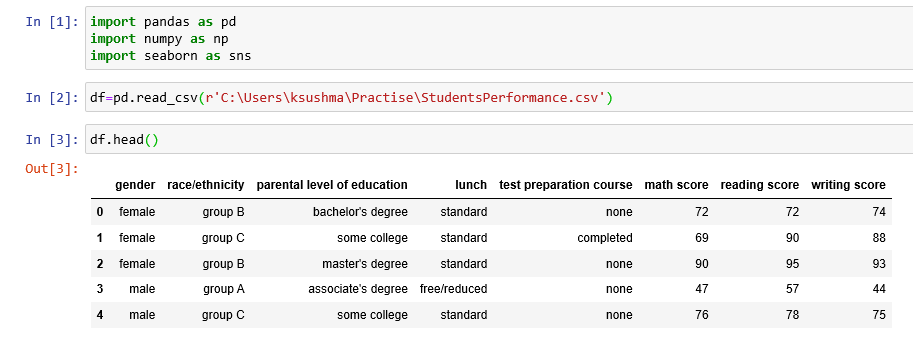
The **n\_jobs** hyperparameter tells the engine how many processors it is allowed to use. If it has a value of one, it can only use one processor. A value of “-1” means that there is no limit.

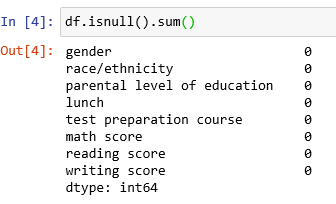
The **random\_state** hyperparameter makes the model’s output replicable. The model will always produce the same results when it has a definite value of random\_state and if it has been given the same hyperparameters and the same training data.

Lastly, there is the **oob\_score** (also called oob sampling), which is a random forest cross-validation method. In this sampling, about one-third of the data is not used to train the model and can be used to evaluate its performance. These samples are called the out-of-bag samples. It's very similar to the leave-one-out-cross-validation method, but almost no additional computational burden goes along with it.

**Example:** Let’s take an example of Student Performance from Kaggle dataset.

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy

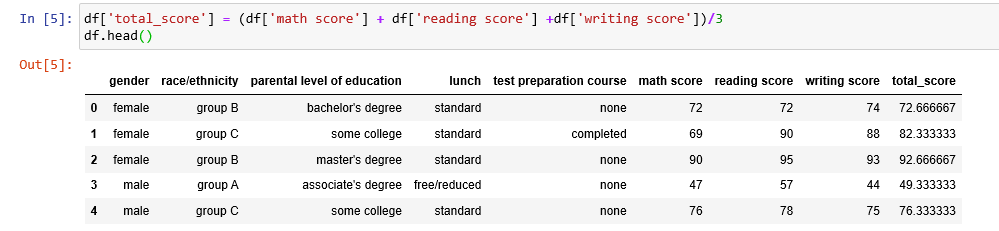




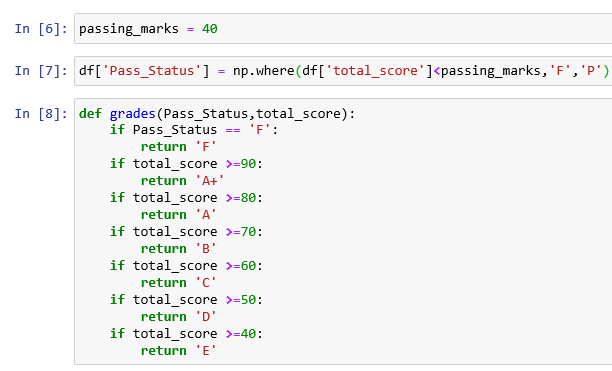
No null values to handle here.

**Feature Scaling**

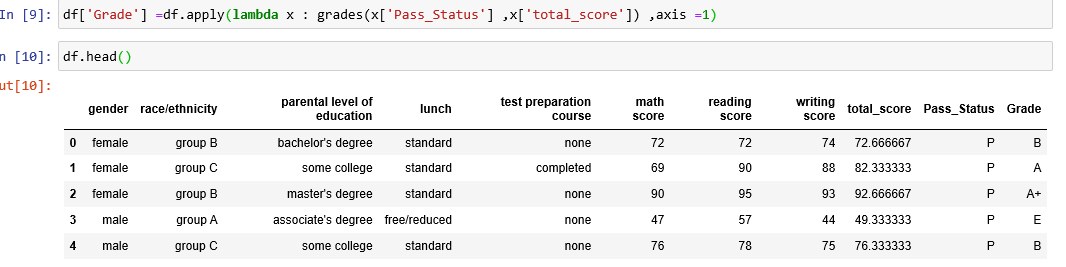
**Step 3:** Handling data with functions used NumPy and Pandas.

Adding Scores together, and assigning them to one variable as total scores.

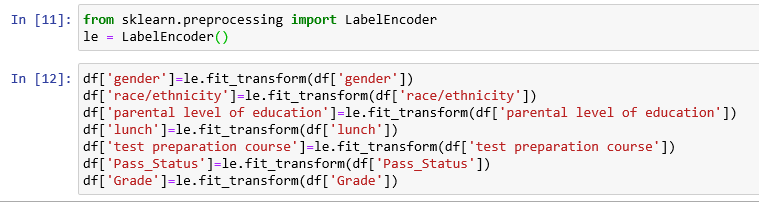
**Step 4:** As per requirement, we need to divide the total scores to pass and fail. For that we are considering pass marks as 40 and also assigning grades as per the Marks.

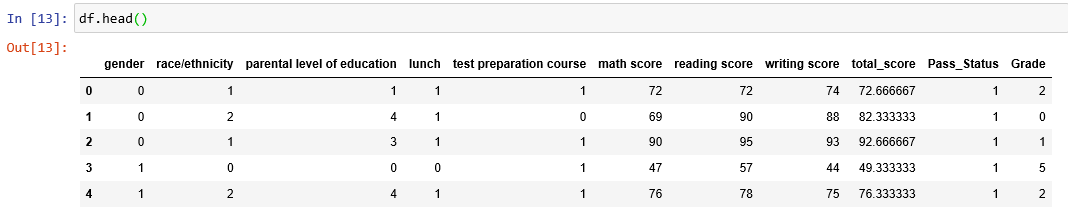


**Step 5:** Adding Grade as one column by using lamda function. So, finally data looks as below.



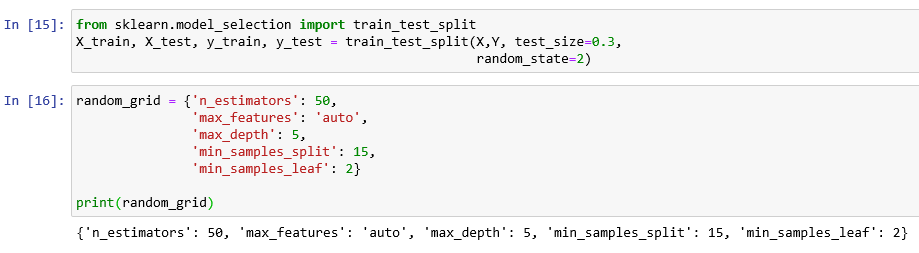
**Step 6:** Converting Categorical Data to Numerical by using Label Encoder.

After encoding, Data is as below.

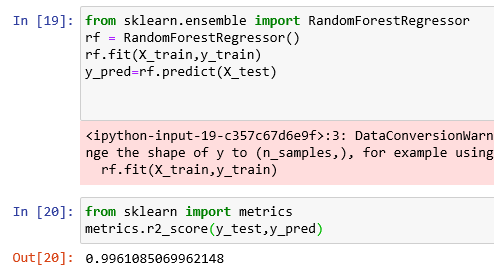
Considering Target variable as total Score and Diving X, Y variables.



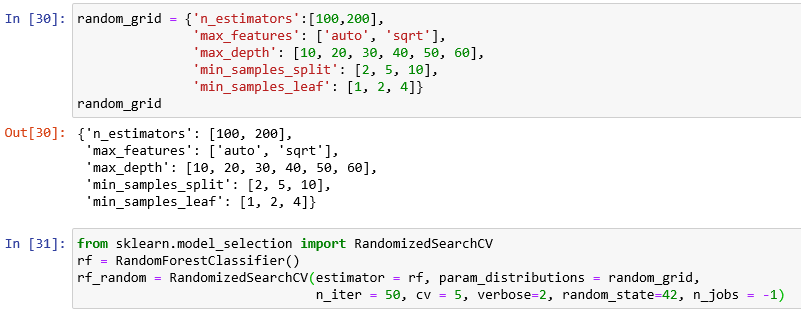
**Step 7:** Train and Test.

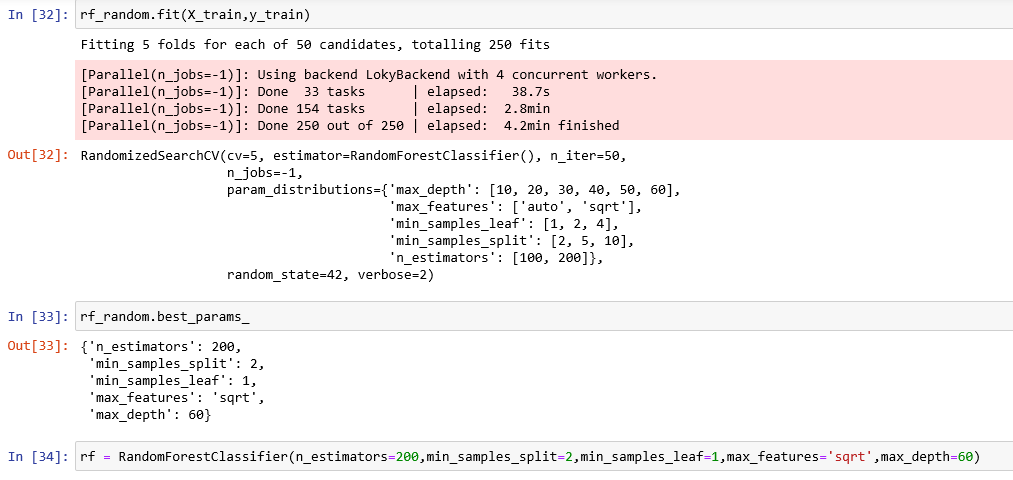
Given parameters in random\_grid and implementing the algorithm.

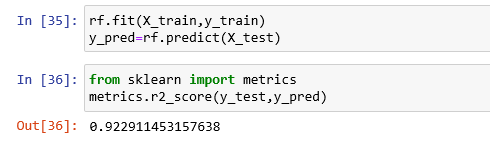
**Step 8:** Implementing Random Forest Algorithm and predicting the accuracy using r2 score.



To find best parameters, using hyper Tuning Parameters.

After fitting train and test data, finding best parameters. Grid Search and Random Search.

Implementing these best params and finding accuracy;



**Random Forest GridsearchCV**

Using Same Parameters for GridsearchCV and finding the best parameters out of them.



Finding best parameters and estimators then implementing with random forest and finding the accuracy.

As, the target Variable is **continuous (total score)**, We are using RandomForestRegressor. If the target Variable is pass status, then we can use RandomForestClassifier.

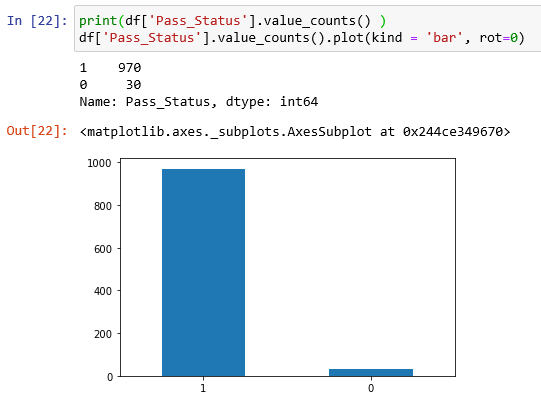
In Random forest Algorithm, we have both Regressor and Classifier.

**Implementing RandomForestClassifier for the same data after feature scaling.**

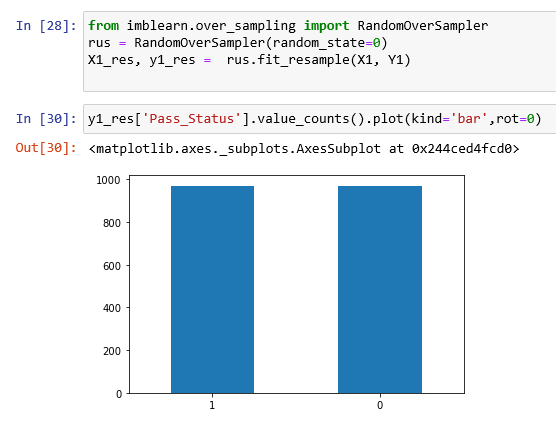
**Step 9:** Defining X and Y by considering target variable as pass status.



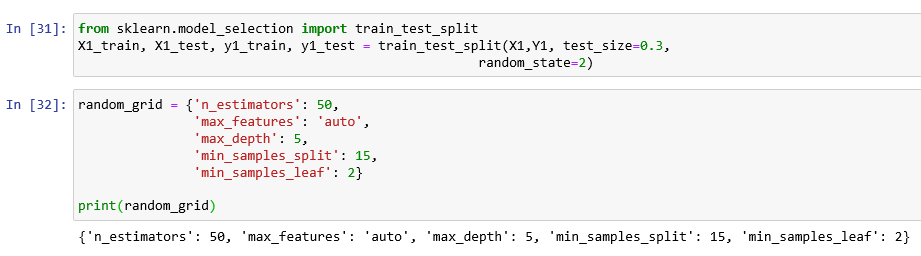
**Step 10:** As the target variable is Categorical, need to check whether the data is balanced or not.



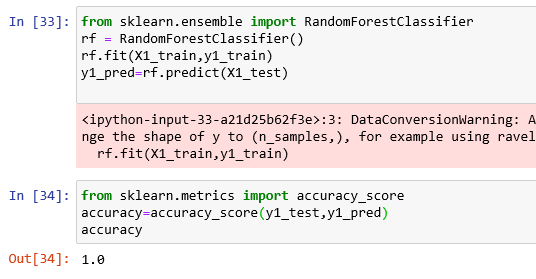
Data is Imbalanced, so we need to balance the data using RamdomOverSampler.



**Step 11:** Train and Test

Given parameters in random\_grid and implementing the algorithm.

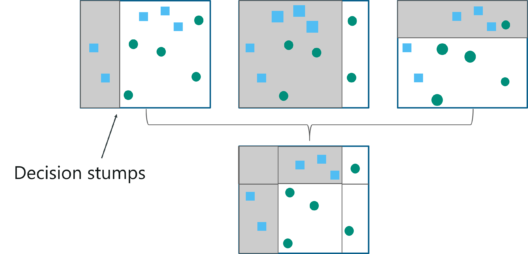
**Step 12:** Implementing RandomClassifier and finding the accuracy.



Ignore Warnings if any. No issues with that.

**Boosting**

The basic principle behind the working of the boosting algorithm is to generate multiple weak learners and combine their predictions to form one strong rule. These weak rules are generated by applying base Machine Learning algorithms on different distributions of the data set. These algorithms generate weak rules for each iteration. After multiple iterations, the weak learners are combined to form a strong learner that will predict a more accurate outcome.



how the algorithm works:

**Step 1:** The base algorithm reads the data and assigns equal weight to each sample observation.

**Step 2:** False predictions made by the base learner are identified. In the next iteration, these false predictions are assigned to the next base learner with a higher weightage on these incorrect predictions.

**Step 3:** Repeat step 2 until the algorithm can correctly classify the output.

Therefore, the main aim of Boosting is to focus more on miss-classified predictions.

Now that we know how the boosting algorithm works, let’s understand the different types of boosting techniques.

## **Adaptive Boosting (Adaboost)**

* AdaBoost is implemented by combining several weak learners into a single strong learner.
* The weak learners in AdaBoost take into account a single input feature and draw out a single split decision tree called the decision stump. Each observation is weighed equally while drawing out the first decision stump.
* The results from the first decision stump are analyzed and if any observations are wrongfully classified, they are assigned higher weights.
* Post this, a new decision stump is drawn by considering the observations with higher weights as more significant.
* Again, if any observations are misclassified, they’re given higher weight and this process continues until all the observations fall into the right class.
* Adaboost can be used for both classification and regression-based problems, however, it is more commonly used for classification purpose.

**Parameters used in Adaboost Regressor.**

1. **base\_estimator object, default=None**

The base estimator from which the boosted ensemble is built. If None, then the base estimator is DecisionTreeRegressor (max\_depth=3).

1. **n\_estimators int, default=50**

The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

1. **learning\_rate float, default=1.**

Learning rate shrinks the contribution of each regressor by learning\_rate. There is a trade-off between learning\_rate and n\_estimators.

1. **loss {‘linear’, ‘square’, ‘exponential’}, default=’linear’**

The loss function to use when updating the weights after each boosting iteration.

1. **random\_state int or Random State, default=None**

Controls the random seed given at each base\_estimator at each boosting iteration. Thus, it is only used when base\_estimator exposes a random\_state. In addition, it controls the bootstrap of the weights used to train the base\_estimator at each boosting iteration. Pass an int for reproducible output across multiple function calls.

**Parameters Used for Adaboost Classifier:**

1. **base\_estimator object, default=None**

The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper classes\_ and n\_classes\_ attributes. If None, then the base estimator is DecisionTreeClassifier (max\_depth=1).

1. **n\_estimators int, default=50**

The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

1. **learning\_rate float, default=1.**

Learning rate shrinks the contribution of each classifier by learning\_rate. There is a trade-off between learning\_rate and n\_estimators.

1. **Algorithm {‘SAMME’, ‘SAMME.R’}, default=’SAMME.R’**

If ‘SAMME.R’ then use the SAMME.R real boosting algorithm. base\_estimator must support calculation of class probabilities. If ‘SAMME’ then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.

1. **random\_state int or Random State, default=None**

Controls the random seed given at each base\_estimator at each boosting iteration. Thus, it is only used when base\_estimator exposes a random\_state. Pass an int for reproducible output across multiple function

1. **Gradient Boosting**

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

## **How Gradient Boosting Works**

Gradient boosting involves three elements:

1. A loss function to be optimized.
2. A weak learner to make predictions.
3. An additive model to add weak learners to minimize the loss function.

### **1. Loss Function**

The loss function used depends on the type of problem being solved. It must be differentiable, but many standard loss functions are supported and you can define your own.

For example, regression may use a squared error and classification may use logarithmic loss. A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

### **2. Weak Learner**

Decision trees are used as the weak learner in gradient boosting. Specifically, regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and “correct” the residuals in the predictions.

Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss. Initially, such as in the case of AdaBoost, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally with 4-to-8 levels. It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes. This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

### **3. Additive Model**

Trees are added one at a time, and existing trees in the model are not changed. A gradient descent procedure is used to minimize the loss when adding trees. Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

Instead of parameters, we have weak learner sub-models or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by (reducing the residual loss. Generally, this approach is called functional gradient descent or gradient descent with functions.

**Parameters Used in Gradient Boosting Regressor:**

**loss {‘ls’, ‘lad’, ‘huber’, ‘quantile’}, default=’ls’:** loss function to be optimized. ‘ls’ refers to least square regression. ‘lad’ (least absolute deviation) is a highly robust loss function solely based on order information of the input variables. ‘huber’ is a combination of the two. ‘quantile’ allows quantile regression (use alpha to specify the quantile).

**learning\_rate float, default=0.1:** learning rate shrinks the contribution of each tree by learning\_rate. There is a trade-off between learning\_rate and n\_estimators.

**n\_estimators int, default=100:** The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

**Subsample float, default=1.0:** The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. subsample interacts with the parameter n\_estimators. Choosing subsample < 1.0 leads to a reduction of variance and an increase in bias.

**criterion {‘friedman\_mse’, ‘mse’, ‘mae’}, default=’friedman\_mse’**

The function to measure the quality of a split. Supported criteria are “friedman\_mse” for the mean squared error with improvement score by Friedman, “mse” for mean squared error, and “mae” for the mean absolute error. The default value of “friedman\_mse” is generally the best as it can provide a better approximation in some cases.

**min\_samples\_split int or float, default=2:** The minimum number of samples required to split an internal node:

* + If int, then consider min\_samples\_split as the minimum number.
  + If float, then min\_samples\_split is a fraction and ceil(min\_samples\_split \* n\_samples) are the minimum number of samples for each split.

**min\_samples\_leaf int or float, default=1:** The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min\_samples\_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

* + If int, then consider min\_samples\_leaf as the minimum number.
  + If float, then min\_samples\_leaf is a fraction and ceil(min\_samples\_leaf \* n\_samples) are the minimum number of samples for each node.

**min\_weight\_fraction\_leaf float, default=0.0:** The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample\_weight is not provided.

**max\_depth int, default=3**: maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

**min\_impurity\_decrease float, default=0.0**: A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

N\_t / N \* (impurity - N\_t\_R / N\_t \* right\_impurity - N\_t\_L / N\_t \* left\_impurity)

where N is the total number of samples, N\_t is the number of samples at the current node, N\_t\_L is the number of samples in the left child, and N\_t\_R is the number of samples in the right child.

N, N\_t, N\_t\_R and N\_t\_L all refer to the weighted sum, if sample\_weight is passed.

**min\_impurity\_split float, default=None:** Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

**initestimator or ‘zero’, default=None:** An estimator object that is used to compute the initial predictions. init has to provide [fit](https://scikit-learn.org/stable/glossary.html#term-fit) and [predict](https://scikit-learn.org/stable/glossary.html#term-predict). If ‘zero’, the initial raw predictions are set to zero. By default, a DummyEstimator is used, predicting either the average target value (for loss=’ls’), or a quantile for the other losses.

**random\_stateint or RandomState, default=None:** Controls the random seed given to each Tree estimator at each boosting iteration. In addition, it controls the random permutation of the features at each split (see Notes for more details). It also controls the random spliting of the training data to obtain a validation set if n\_iter\_no\_change is not None. Pass an int for reproducible output across multiple function calls.

**max\_features{‘auto’, ‘sqrt’, ‘log2’}, int or float, default=None:** The number of features to consider when looking for the best split:

* + If int, then consider max\_features features at each split.
  + If float, then max\_features is a fraction and int(max\_features \* n\_features) features are considered at each split.
  + If “auto”, then max\_features=n\_features.
  + If “sqrt”, then max\_features=sqrt(n\_features).
  + If “log2”, then max\_features=log2(n\_features).
  + If None, then max\_features=n\_features.
* Choosing max\_features < n\_features leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max\_features features.

**Alpha float, default=0.9**: The alpha-quantile of the huber loss function and the quantile loss function. Only if loss='huber' or loss='quantile'.

**Verbose int, default=0:** Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

**max\_leaf\_nodes int, default=None:** Grow trees with max\_leaf\_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

**warm\_start bool, default=False:** When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution. **validation\_fraction float, default=0.1:** The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if n\_iter\_no\_change is set to an integer.

**n\_iter\_no\_change int, default=None**

n\_iter\_no\_change is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside validation\_fraction size of the training data as validation and terminate training when validation score is not improving in all of the previous n\_iter\_no\_change numbers of iteration

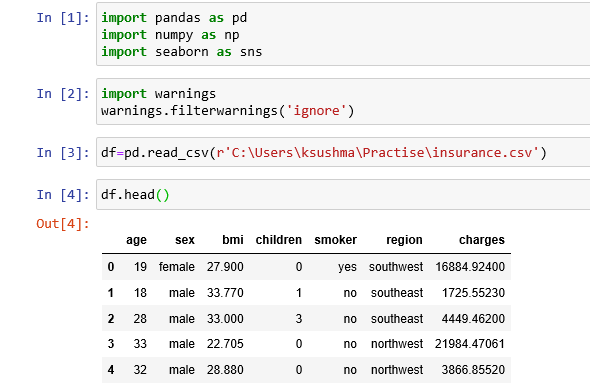
**Tol float, default=1e-4:** Tolerance for the early stopping. When the loss is not improving by at least tol for n\_iter\_no\_change iterations (if set to a number), the training stops.

**ccp\_alpha non-negative float, default=0.0:** Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp\_alpha will be chosen. By default, no pruning is performed.

**Adaboost and Gradient boost Example for Regression.**

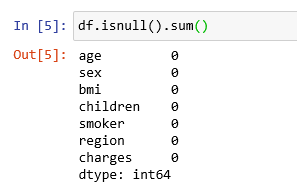
**Example:** Let's take an example of Medical Insurance Dataset from Kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy



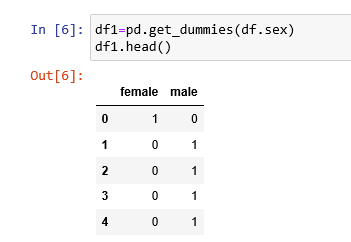
**EDA Analysis**

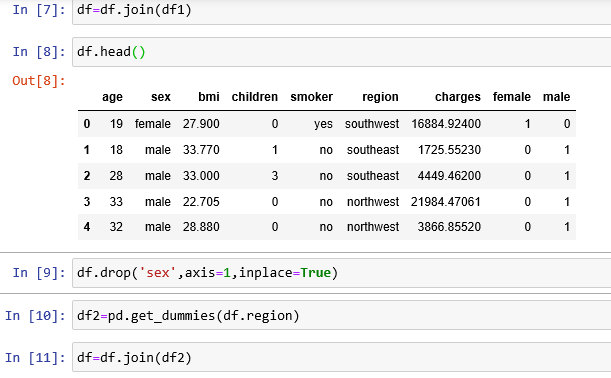
**Step 2:** Finding Null Values if any;

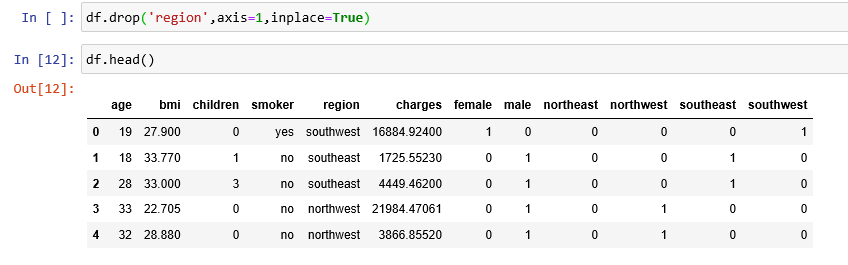


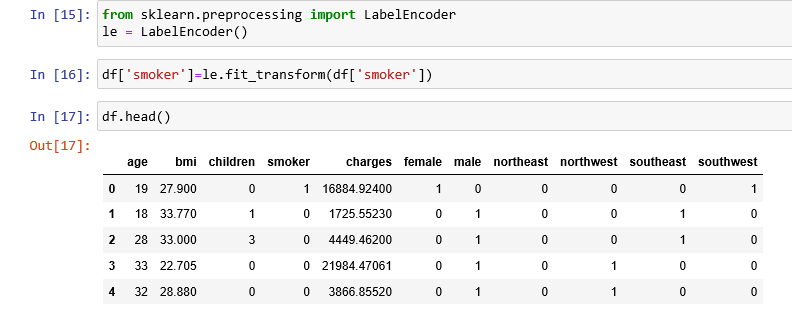
**Feature Engineerinng**

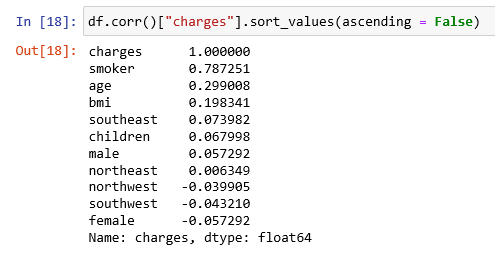
**Step 3:** One-hot Encoding for S**ex and Region** column from the given dataset.



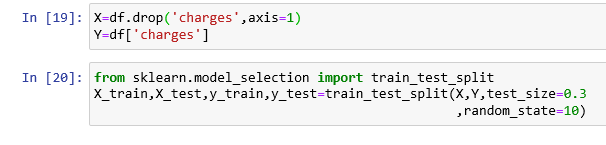


Final data after feature Engineering Analysis.**Step 4:** Converting Categorical Data to Numerical Data.

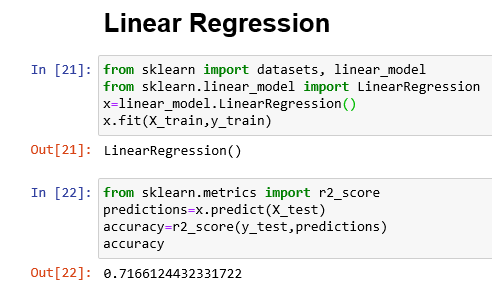
**Step 5:** Finding the correlation.



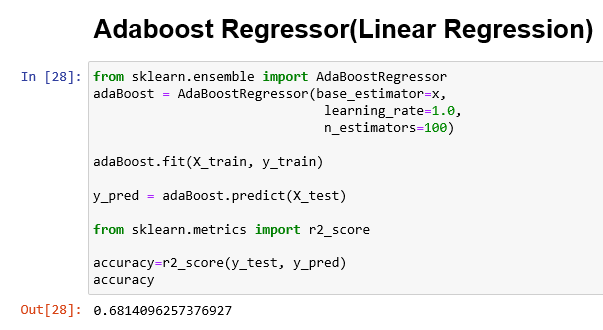
**Step 6:** Defining X , Y values and Splitting data to Train and test data.



**Step 7:** Perform Algorithms by importing specific packages

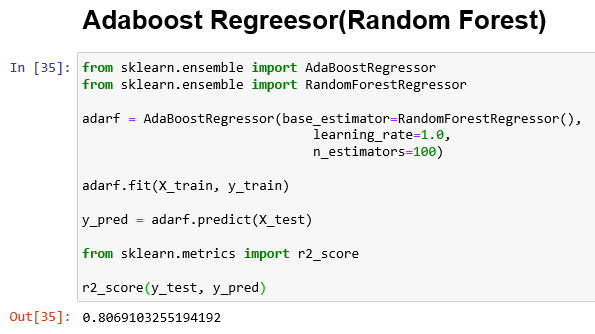
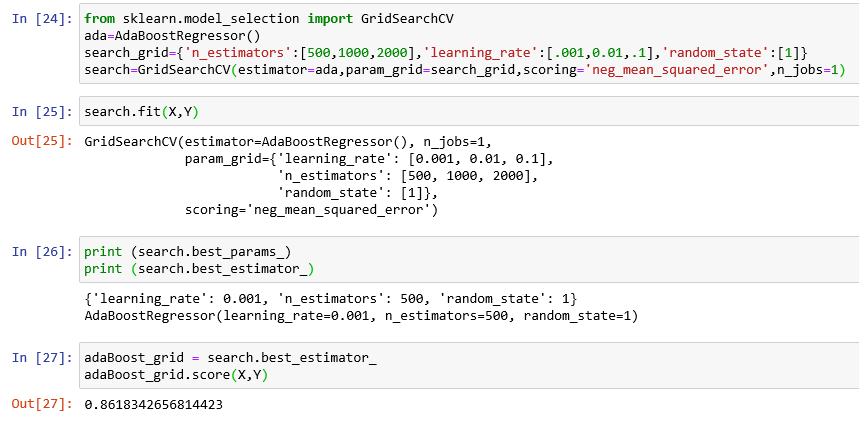
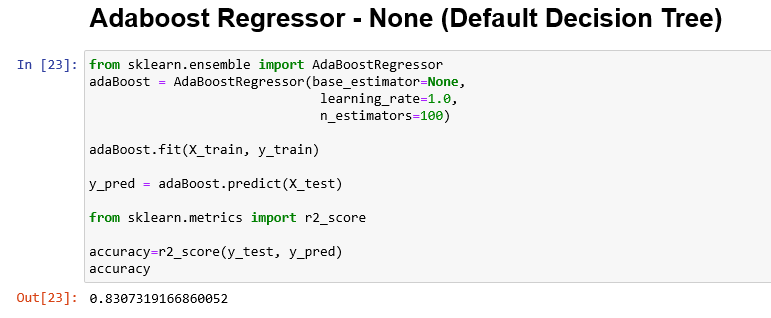


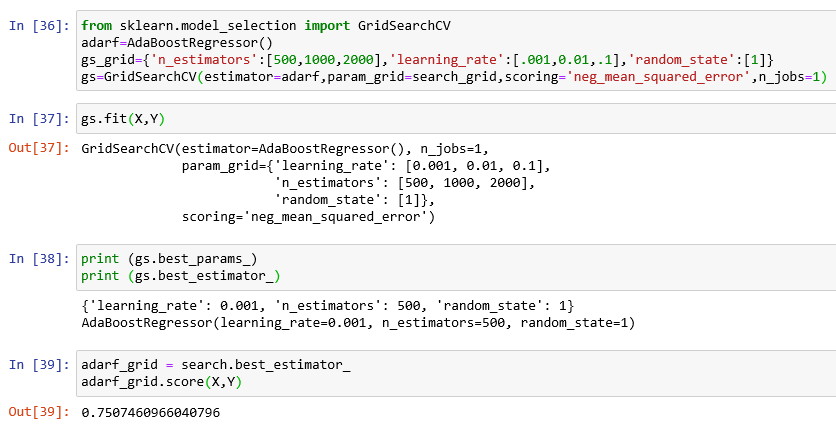
Applying Adaboost Regressor with GridSearchCV.



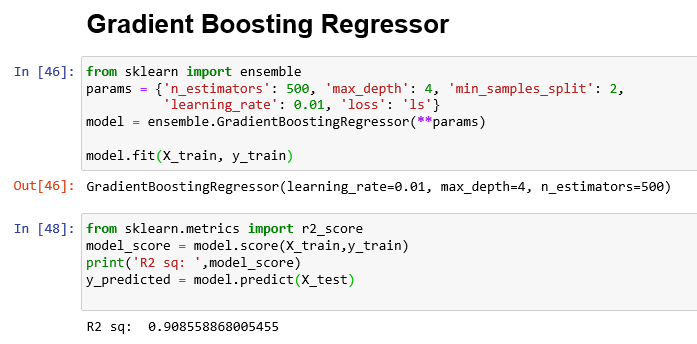


Adaboost Regressor with Default Decision Tree





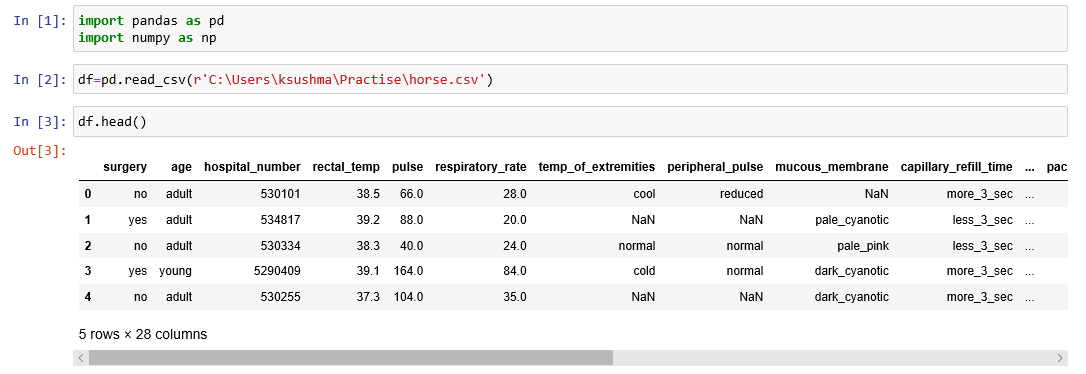
Perform Adaboost Regressor with all the Algorithms which suits the data (either Regression or Classification) and Compare the accuracy.



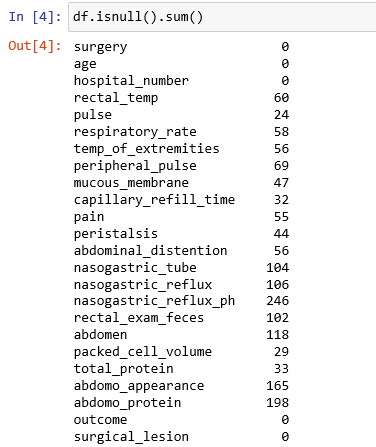
**Adaboost and Gradient Boost for Classification Algorithms.**

**Example:** Let's take an example of Medical Insurance Dataset from Kaggle.com

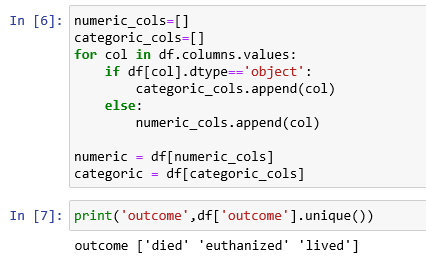
**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy

**EDA Analysis:**

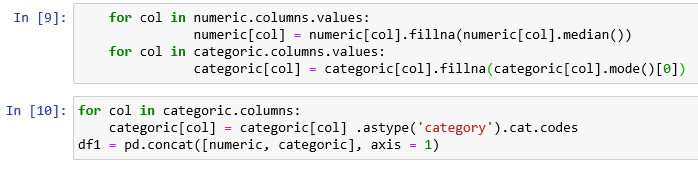
**Step 2:** Checking for null values if any;



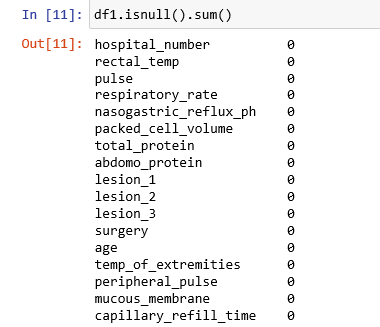
**Step 3:** Dividing Categorical and Numerical Values



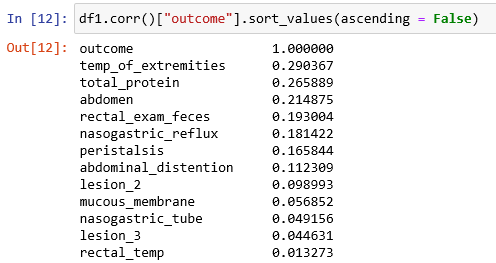
**Step 4:** Replacing Numeric Values with median and categoric with mode and Concatenating both.



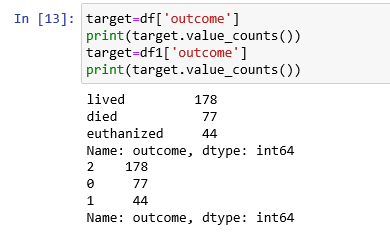
Validating Null values again after replacing null Values.



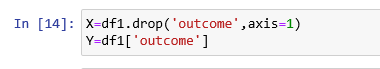
**Step 5:** Finding the Correlation



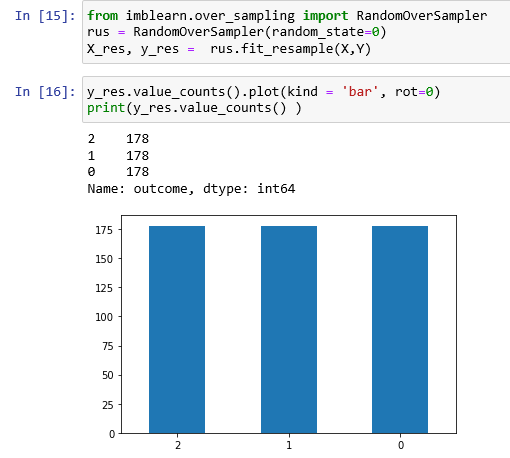
Step 6: Checking Target Values for Imbalanced Data.



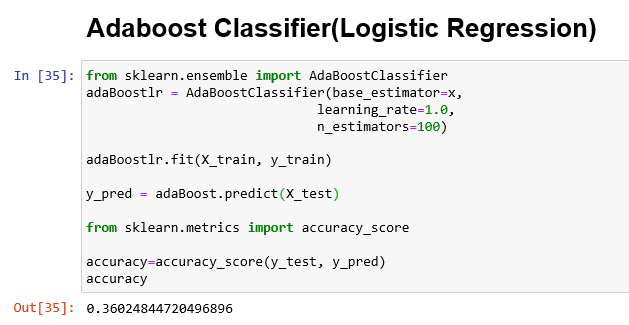
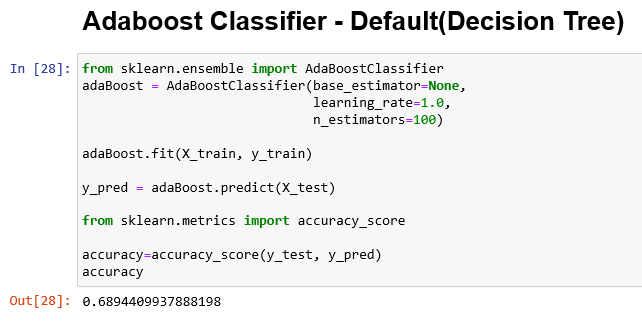
**Step 7:** Dividing X and Y Values.

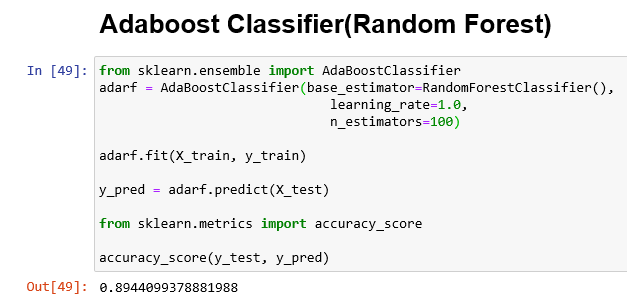
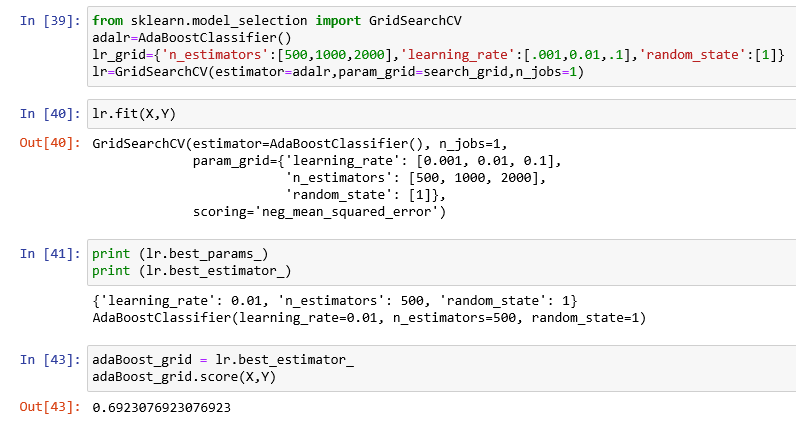


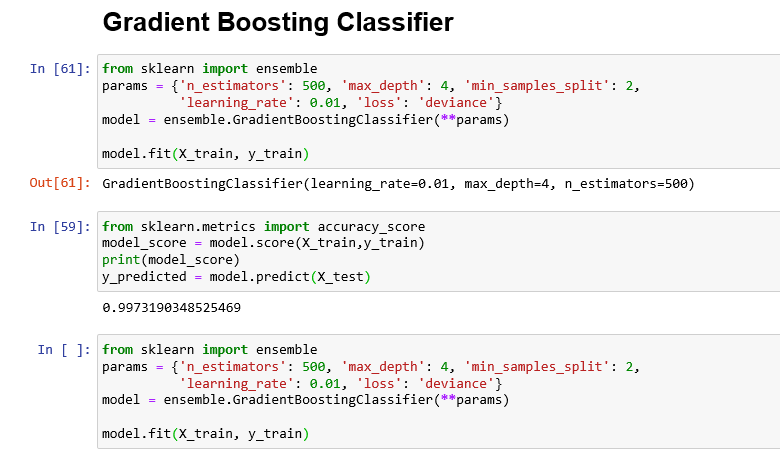
**Step 8:** Balancing the data using RandomOverSampler



**Step 9:** Train and Split the data







**XGBoost**

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

XGBoost is a software library that you can download and install on your machine, then access from a variety of interfaces. Specifically, XGBoost supports the following main interfaces:

* Command Line Interface (CLI).
* C++ (the language in which the library is written).
* Python interface as well as a model in scikit-learn.
* R interface as well as a model in the caret package.
* Julia.
* Java and JVM languages like Scala and platforms like Hadoop.

## **XGBoost Features**

The library is laser focused on computational speed and model performance, as such there are few frills. Nevertheless, it does offer a number of advanced features.

### **Model Features**

The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

* **Gradient Boosting:** algorithm also called gradient boosting machine including the learning rate.
* **Stochastic Gradient Boosting:** with sub-sampling at the row, column and column per split levels.
* **Regularized Gradient Boosting:** with both L1 and L2 regularization.

### **System Features**

The library provides a system for use in a range of computing environments, not least:

* **Parallelization** of tree construction using all of your CPU cores during training.
* **Distributed Computing** for training very large models using a cluster of machines.
* **Out-of-Core Computing** for very large datasets that don’t fit into memory.
* **Cache Optimization** of data structures and algorithm to make best use of hardware.

### **Algorithm Features**

The implementation of the algorithm was engineered for efficiency of compute time and memory resources. A design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

* **Sparse Aware** implementation with automatic handling of missing data values.
* **Block Structure** to support the parallelization of tree construction.
* **Continued Training** so that you can further boost an already fitted model on new data.

## **Why Use XGBoost?**

The two reasons to use XGBoost are also the two goals of the project:

1. Execution Speed.
2. Model Performance.

### **1. XGBoost Execution Speed**

Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting.

### **2. XGBoost Model Performance**

XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems.

The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform.

## **What Algorithm Does XGBoost Use?**

The XGBoost library implements the [gradient boosting decision tree algorithm](https://en.wikipedia.org/wiki/Gradient_boosting).

This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.

Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. A popular example is the [AdaBoost algorithm](http://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/) that weights data points that are hard to predict.

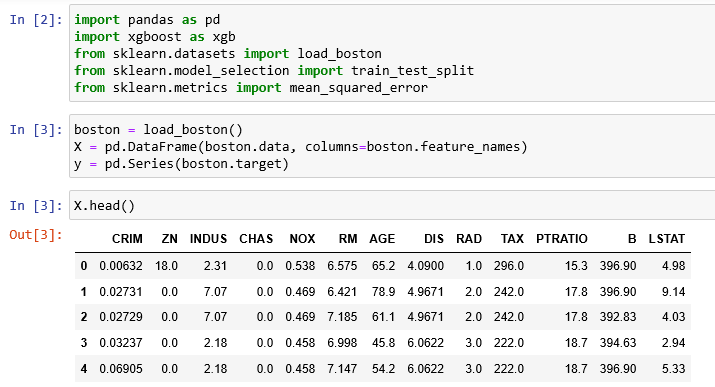
Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

This approach supports both regression and classification predictive modeling problems.

**Example:** Let's take an example of Boston dataset from sktlearn.

**Step 1:** Importing all packages which were needed to execute xgboost algorithm.

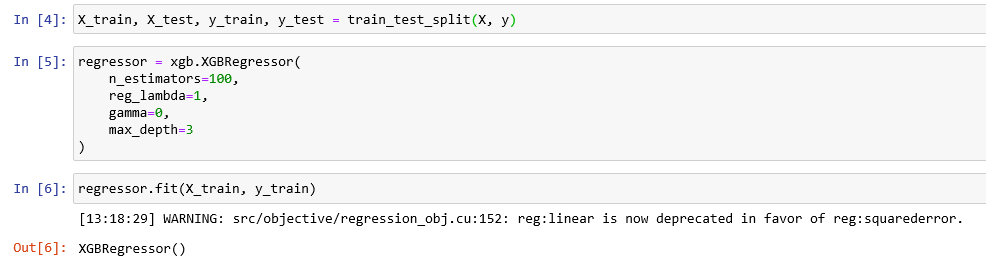
Importing data from csv file by providing path and imported libraries pandas and NumPy

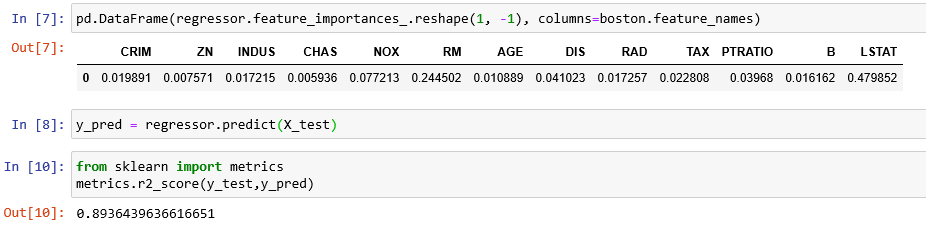


**Step 2:** **Train and Test**

No null values found from the data set and data is fitted to train and split

Applying XGBoostregressor().

**Step 3:** Creating data frame for the required columns and finding the aacuracy.



**K-Means**

## **What is Clustering?**

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

Clustering is the process of dividing the entire data into groups (also known as clusters) based on the patterns in the data.

Example: A bank wants to give credit card offers to its customers. Currently, they look at the details of each customer and based on this information, decide which offer should be given to which customer. Now, the bank can potentially have millions of customers. Does it make sense to look at the details of each customer separately and then make a decision? Certainly not! It is a manual process and will take a huge amount of time.

So, what can the bank do? One option is to segment its customers into different groups. For instance, the bank can group the customers based on their income:

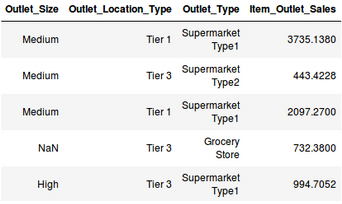
**High Income Average Income Low Income**

The bank can now make three different strategies or offers, one for each group. Here, instead of creating different strategies for individual customers, they only have to make 3 strategies. This will reduce the effort as well as the time.

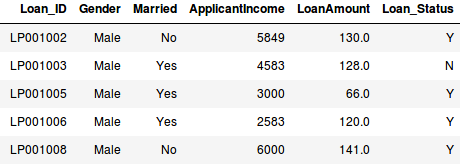
The groups which shown above are known as clusters and the process of creating these groups is known as clustering.

## **How is Clustering an Unsupervised Learning Problem?**

Let’s say you are working on a project where you need to predict the sales of a big mart:



Or, a project where your task is to predict whether a loan will be approved or not:

  
We have a fixed target to predict in both of these situations. In the sales prediction problem, we have to predict the Item\_Outlet\_Sales based on outlet\_size, outlet\_location\_type, etc. and in the loan approval problem, we have to predict the Loan\_Status depending on the Gender, marital status, the income of the customers, etc.

So, when we have a target variable to predict based on a given set of predictors or independent variables, such problems are called supervised learning problems.

Now, there might be situations where we do not have any target variable to predict.

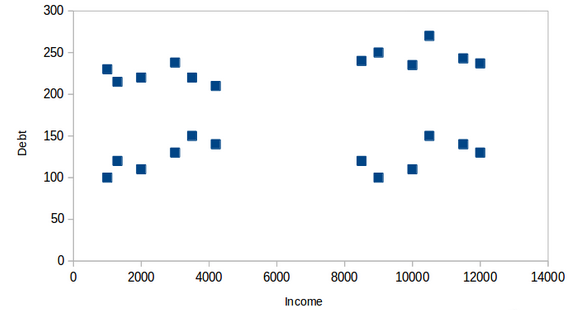
Such problems, without any fixed target variable, are known as unsupervised learning problems. In these problems, we only have the independent variables and no target/dependent variable.

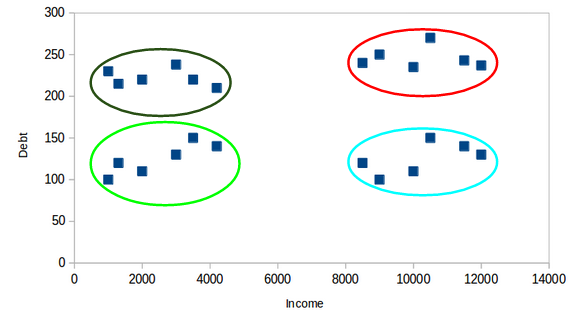
In clustering, we do not have a target to predict. We look at the data and then try to club similar observations and form different groups. Hence it is an unsupervised learning problem.

We now know what are clusters and the concept of clustering. Next, let’s look at the properties of these clusters which we must consider while forming the clusters.

## **Properties of Clusters**

How about another example? We’ll take the same bank as before who wants to segment its customers. For simplicity purposes, let’s say the bank only wants to use the income and debt to make the segmentation. They collected the customer data and used a scatter plot to visualize it:

  
On the X-axis, we have the income of the customer and the y-axis represents the amount of debt. Here, we can clearly visualize that these customers can be segmented into 4 different clusters as shown below:

  
This is how clustering helps to create segments (clusters) from the data. The bank can further use these clusters to make strategies and offer discounts to its customers.

## **Applications of Clustering in Real-World Scenarios**

Clustering is a widely used technique in the industry. It is actually being used in almost every domain, ranging from banking to recommendation engines, document clustering to image segmentation.

### **Customer Segmentation**

We covered this earlier – one of the most common applications of clustering is customer segmentation. And it isn’t just limited to banking. This strategy is across functions, including telecom, e-commerce, sports, advertising, sales, etc.

### **Document Clustering**

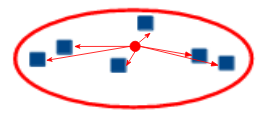
This is another common application of clustering. Let’s say you have multiple documents and you need to cluster similar documents together. Clustering helps us group these documents such that similar documents are in the same clusters.  
 **3. Image Segmentation**

We can also use clustering to perform image segmentation. Here, we try to club similar pixels in the image together. We can apply clustering to create clusters having similar pixels in the same group.

### **Inertia**

Recall the first property of clusters we covered above. This is what inertia evaluates. It tells us how far the points within a cluster are. So, inertia actually calculates the sum of distances of all the points within a cluster from the centroid of that cluster.

We calculate this for all the clusters and the final inertial value is the sum of all these distances. This distance within the clusters is known as intracluster distance. So, inertia gives us the sum of intracluster distances:



Now, what do you think should be the value of inertia for a good cluster? Is a small inertial value good or do we need a larger value? We want the points within the same cluster to be similar to each other, right? Hence, the distance between them should be as low as possible.

## **Introduction to K-Means Clustering**

Recall the first property of clusters – it states that the points within a cluster should be similar to each other. So, our aim here is to minimize the distance between the points within a cluster.

There is an algorithm that tries to minimize the distance of the points in a cluster with their centroid – the k-means clustering technique. K-means is a centroid-based algorithm, or a distance-based algorithm, where we calculate the distances to assign a point to a cluster. In K-Means, each cluster is associated with a centroid. The main objective of the K-Means algorithm is to minimize the sum of distances between the points and their respective cluster centroid.

Let’s now take an example to understand how K-Means actually works:

### **Step 1: Choose the number of clusters *k***

The first step in k-means is to pick the number of clusters, k

### **Step 2: Select k random points from the data as centroids**

Next, we randomly select the centroid for each cluster. Let’s say we want to have 2 clusters, so k is equal to 2 here. We then randomly select the centroid:

### **Step 3: Assign all the points to the closest cluster centroid**

Once we have initialized the centroids, we assign each point to the closest cluster centroid

### **Step 4: Recompute the centroids of newly formed clusters**

Now, once we have assigned all of the points to either cluster, the next step is to compute the centroids of newly formed clusters

### **Step 5: Repeat steps 3 and 4**

We then repeat steps 3 and 4:

The step of computing the centroid and assigning all the points to the cluster based on their distance from the centroid is a single iteration.

### **Stopping Criteria for K-Means Clustering**

There are essentially three stopping criteria that can be adopted to stop the K-means algorithm:

* Centroids of newly formed clusters do not change
* Points remain in the same cluster
* Maximum number of iterations are reached

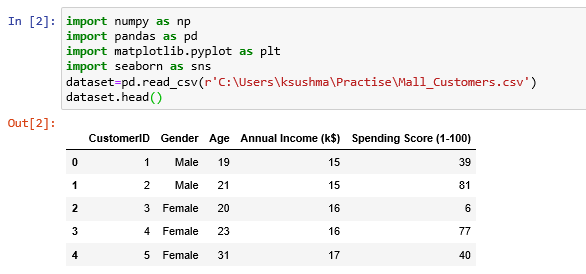
We can stop the algorithm if the centroids of newly formed clusters are not changing. Even after multiple iterations, if we are getting the same centroids for all the clusters, we can say that the algorithm is not learning any new pattern and it is a sign to stop the training.

Another clear sign that we should stop the training process if the points remain in the same cluster even after training the algorithm for multiple iterations.

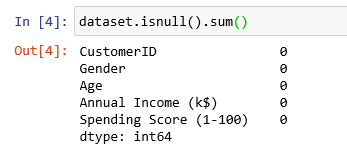
Finally, we can stop the training if the maximum number of iterations is reached. Suppose if we have set the number of iterations as 100. The process will repeat for 100 iterations before stopping.

**Example:** Let's take an example of customers dataset from kaggle.com

**Step 1:** Importing data from csv file by providing path and imported libraries pandas and NumPy



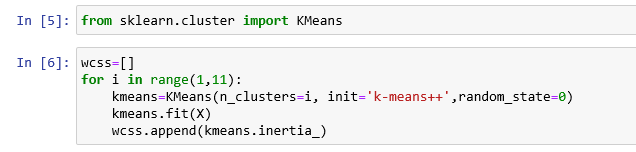
**EDA Analysis**

**Step 2:** Checking for Null Values if any;  


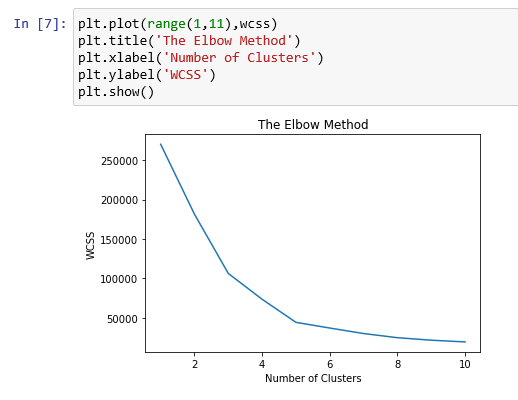
**Step 3:** Assigning X values by using iloc (positions).



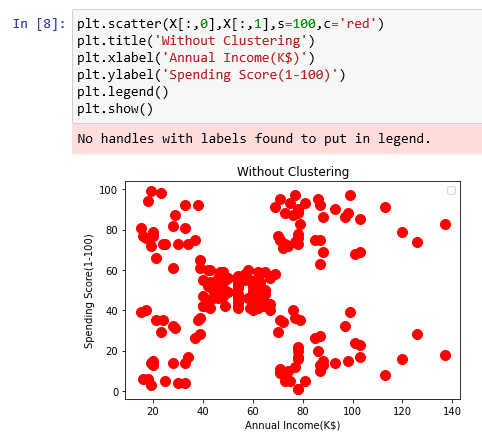
**Step 4:** importing kmeans and parameters which needed



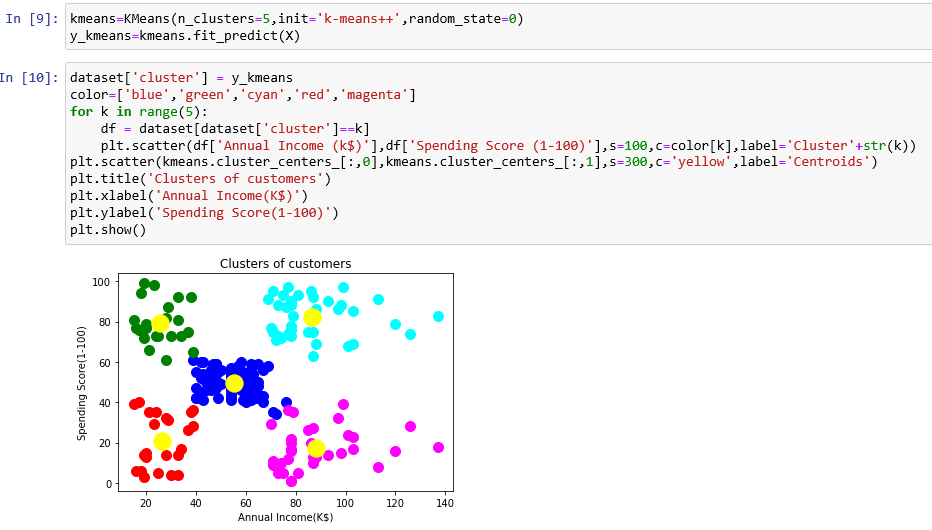
**Step 5:** Showing the elbow method in pictorial format.



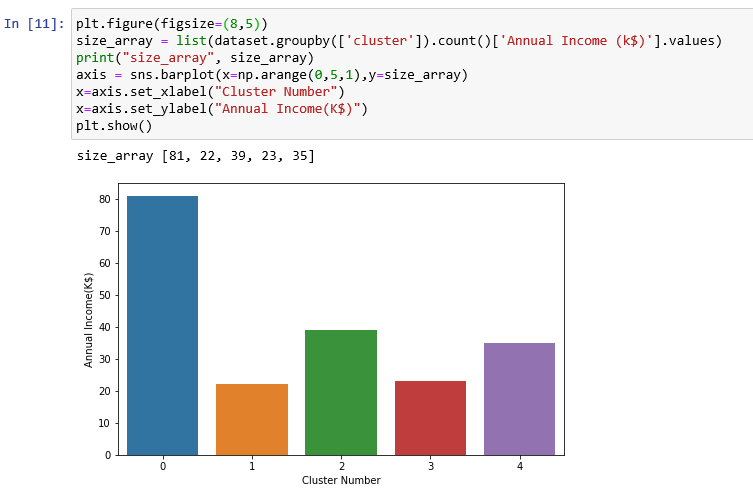
**Step 6:** Pictorial format without clustering the data.



**Step 7:** Applying kmeans++ and clusters, then showing clusters among datapoints in pictorial format.



**Step8:** Assigning numbers to clusters and showing them in bar graph.



**Step 9:** Representing the data, that which cluster is assigned to which kind of data.

